

APIs Overview

This section provides documents on various APIs.

Title	Description
AvalancheGo	AvalancheGo APIs, Postman collection, public servers, and release notes
AvalancheJS	AvalancheJS APIs
Avalanche Metrics API	Link to Avalanche Metrics API
Glacier API	High-performance API for easy access to indexed blockchain data

AvalancheGo Overview

[APIs](#): all APIs provided by AvalancheGo server software.

[Public API Server](#): the public API server you can use to interact with the Avalanche Network.

[Postman Collection](#): an AvalancheGo API template and environment collection for [Postman](#), a tool to simplify API development and experimentation. The collection covers all of AvalancheGo APIs, so you can easily issue commands to a node and examine the output in a nicely formatted, readable way.

Release Notes: [the latest release notes on GitHub](#).

sidebar_position: 1

AvalancheGo APIs Overview

Title	Description
Issuing API Calls	This guide explains how to make calls to APIs exposed by Avalanche nodes.
The Platform Chain (P-Chain) API	Allows clients to interact with the P-Chain (Platform Chain), which maintains Avalanche's validator set and handles blockchain and Subnet creation.
The Contract Chain (C-Chain) API	Allows clients to interact with the C-Chain, Avalanche's main EVM instance, as well as other EVM instances.
The Exchange Chain (X-Chain) API	Allows clients to create and trade assets, including AVAX, on the X-Chain as well as other instances of the AVM.
The Admin API	Allows clients to examine a node's internal state, set of connections, and similar internal protocol data.
The Auth API	Allows clients to manage the creation and revocation of authorization tokens.
The Health API	Allows clients to check a node's health.
The Index API	Fetch transactions, vertex, or block by ID.
The Info API	Allows clients to examine basic information about a node.
The IPC API	Allows users to create Unix domain sockets for blockchains to publish to.
The Keystore API	Allows customers to use the embedded Keystore file of an Avalanche node.
The Metrics API	Allows clients to get statistics about a node's health and performance.

sidebar_position: 6

Admin API

This API can be used for measuring node health and debugging.

:::info The Admin API is disabled by default for security reasons. To run a node with the Admin API enabled, use [config flag](#) `--api-admin-enabled=true`.

This API set is for a specific node, it is unavailable on the [public server](#).

:::

Format

This API uses the `json 2.0` RPC format. For details, see [here](#).

Endpoint

```
/ext/admin
```

Methods

admin.alias

Assign an API endpoint an alias, a different endpoint for the API. The original endpoint will still work. This change only affects this node; other nodes will not know about this alias.

Signature:

```
admin.alias({endpoint:string, alias:string}) -> {}
```

- `endpoint` is the original endpoint of the API. `endpoint` should only include the part of the endpoint after `/ext/`.
- The API being aliased can now be called at `ext/alias`.
- `alias` can be at most 512 characters.

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "method": "admin.alias",  
    "params": {  
        "alias": "myAlias",  
        "endpoint": "bc/X"  
    }  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {}  
}
```

Now, calls to the X-Chain can be made to either `/ext/bc/X` or, equivalently, to `/ext/myAlias`.

admin.aliasChain

Give a blockchain an alias, a different name that can be used anywhere the blockchain's ID is used.

:::note Aliasing a chain can also be done via the [Node API](#). Note that the alias is set for each chain on each node individually. In a multi-node Subnet, the same alias should be configured on each node to use an alias across a Subnet successfully. Setting an alias for a chain on one node does not register that alias with other nodes automatically.

...

Signature:

```
admin.aliasChain(  
    {  
        chain:string,  
        alias:string  
    }  
) -> {}
```

- `chain` is the blockchain's ID.
- `alias` can now be used in place of the blockchain's ID (in API endpoints, for example.)

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "method": "admin.aliasChain",  
    "params": {  
        "chain": "sV6o671RtkGBcnolFiaDbVcFv2sG5aVXMZYzKdP4VQAWmJQnM",  
        "alias": "myBlockchainAlias"  
    }  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

Now, instead of interacting with the blockchain whose ID is `sV6o671RtkGBcn01FiaDbVcFv2sG5aVXMZYzKdP4VQAWmJQnM` by making API calls to `/ext/bc/sV6o671RtkGBcn01FiaDbVcFv2sG5aVXMZYzKdP4VQAWmJQnM`, one can also make calls to `ext/bc/myBlockchainAlias`.

`admin.getChainAliases`

Returns the aliases of the chain

Signature:

```
admin.getChainAliases(
  {
    chain:string
  }
) -> (aliases:string[])
```

- `chain` is the blockchain's ID.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.getChainAliases",
  "params": {
    "chain": "sV6o671RtkGBcn01FiaDbVcFv2sG5aVXMZYzKdP4VQAWmJQnM"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "aliases": [
      "X",
      "avm",
      "2eNy1mUFdmaxXNj1eQUe7Np4gju9sJsEtWQ4MX3ToiNKuADed"
    ],
    "id": 1
  }
}
```

`admin.getLoggerLevel`

Returns log and display levels of loggers.

Signature:

```
admin.getLoggerLevel(
  {
    loggerName:string // optional
  }
) -> {
  loggerLevels: {
    loggerName: {
      logLevel: string,
      displayLevel: string
    }
  }
}
```

- `loggerName` is the name of the logger to be returned. This is an optional argument. If not specified, it returns all possible loggers.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.getLoggerLevel",
  "params": {

```

```
        "loggerName": "C"
    }
}
} ' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "loggerLevels": {
      "C": {
        "logLevel": "DEBUG",
        "displayLevel": "INFO"
      }
    }
  },
  "id": 1
}
```

admin.loadVMs

Dynamically loads any virtual machines installed on the node as plugins. See [here](#) for more information on how to install a virtual machine on a node.

Signature:

```
admin.loadVMs() -> {
  newVMs: map[string][]string
  failedVMs: map[string]string,
}
```

- `failedVMs` is only included in the response if at least one virtual machine fails to be loaded.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.loadVMs",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "newVMs": {
      "tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH": ["foovm"]
    },
    "failedVMs": {
      "rXJsCSEYXg2TehWxCEEGj6JU2PWKTkd6cBdNLjoe2SpSKD9cy": "error message"
    }
  },
  "id": 1
}
```

admin.lockProfile

Writes a profile of mutex statistics to `lock.profile`.

Signature:

```
admin.lockProfile() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.lockProfile",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

admin.memoryProfile

Writes a memory profile of the to `mem.profile`.

Signature:

```
admin.memoryProfile() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.memoryProfile",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

admin.setLoggerLevel

Sets log and display levels of loggers.

Signature:

```
admin.setLoggerLevel(
  {
    loggerName: string, // optional
    logLevel: string, // optional
    displayLevel: string, // optional
  }
) -> {}
```

- `loggerName` is the logger's name to be changed. This is an optional parameter. If not specified, it changes all possible loggers.
- `logLevel` is the log level of written logs, can be omitted.
- `displayLevel` is the log level of displayed logs, can be omitted.

`logLevel` and `displayLevel` cannot be omitted at the same time.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.setLoggerLevel",
  "params": {
    "loggerName": "C",
    "logLevel": "DEBUG",
    "displayLevel": "INFO"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

admin.startCPUProfiler

Start profiling the CPU utilization of the node. To stop, call `admin.stopCPUProfiler`. On stop, writes the profile to `cpu.profile`.

Signature:

```
admin.startCPUProfiler() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.startCPUProfiler",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

admin.stopCPUProfiler

Stop the CPU profile that was previously started.

Signature:

```
admin.stopCPUProfiler() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.stopCPUProfiler"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

sidebar_position: 7

Auth API

When you run a node, you can require that API calls have an authorization token attached. This API manages the creation and revocation of authorization tokens.

An authorization token provides access to one or more API endpoints. This is useful for delegating access to a node's APIs. Tokens expire after 12 hours.

An authorization token is provided in the header of an API call. Specifically, the header `Authorization` should have value `Bearer TOKEN.GOES.HERE` (where `TOKEN.GOES.HERE` is replaced with the token).

This API is only reachable if the node is started with `config.flag --api-auth-required`. If the node is started without this CLI, API calls do not require authorization tokens, so this API is not reachable. This API never requires an authorization token to be reached.

Authorization token creation must be permissioned. If you run your node with `--api-auth-required`, you must also specify the file to read the Auth API's password from, with argument `--api-auth-password-file`. You must provide this password in order to create/revoke authorization tokens.

:::info If you run your node with `--api-auth-required` then some tools like MetaMask may not be able to make API calls to your node because they don't have an auth token.

This API set is for a specific node, it is unavailable on the [public server](#).

:::

Format

This API uses the `json 2.0` RPC format. For more information on making JSON RPC calls, see [here](#).

Endpoint

```
/ext/auth
```

Methods

auth.newToken

Creates a new authorization token that grants access to one or more API endpoints.

Signature:

```
auth.newToken(  
  {  
    password: string,  
    endpoints: []string  
  }  
) -> {token: string}
```

- `password` is this node's authorization token password.
- `endpoints` is a list of endpoints that will be accessible using the generated token. If `endpoints` contains an element `"*"`, the generated token can access any API endpoint.
- `token` is the authorization token.

Example Call:

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "method": "auth.newToken",  
  "params":{  
    "password":"YOUR PASSWORD GOES HERE",  
    "endpoints":["/ext/bc/X", "/ext/info"]  
  },  
  "id": 1  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/auth
```

This call will generate an authorization token that allows access to API endpoints `/ext/bc/X` (that is the X-Chain) and `/ext/info` (that is the [info API](#)).

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJFbmRwb2ludHMiOlsikJdLCJleHAiOjE1OTM0NzU4OTR9.Cqo7TraN_CFN13q3ae4GRJCMgd8Z01QwBzyC29M6Aps",  
  },  
  "id": 1  
}
```

This authorization token should be included in API calls by giving header `Authorization` value `Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJFbmRwb2ludHMiOlsikJdLCJleHAiOjE1OTM0NzU4OTR9.Cqo7TraN_CFN13q3ae4GRJCMgd8Z01QwBzyC29M6Aps`.

For example, to call `info.peers` with this token:

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "info.peers"  
' 127.0.0.1:9650/ext/info \  
-H 'content-type:application/json;' \  
-H 'Authorization: Bearer  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJFbmRwb2ludHMiOlsikJdLCJleHAiOjE1OTM0NzU4OTR9.Cqo7TraN_CFN13q3ae4GRJCMgd8Z01QwBzyC29M6Aps'
```

auth.revokeToken

Revoke a previously generated token. The given token will no longer grant access to any endpoint. If the token is invalid, does nothing.

Signature:

```
auth.revokeToken(  
  {  
    password: string,  
    token: string  
  }  
) -> {}
```

- `password` is this node's authorization token password.

- `token` is the authorization token being revoked.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "auth.revokeToken",
  "params": {
    "password": "123",
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOiE1OTMxNzIzMjh9.qZVNhH6AMQ_LpbXnPbTFEL6Vm5EM5FLU-VEKpYBH3k4"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/auth
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {},
  "id": 1
}
```

auth.changePassword

Change this node's authorization token password. Any authorization tokens created under an old password will become invalid.

Signature:

```
auth.changePassword(
  {
    oldPassword: string,
    newPassword: string
  }
) -> {}
```

- `oldPassword` is this node's current authorization token password.
- `newPassword` is the node's new authorization token password after this API call. Must be between 1 and 1024 characters.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "auth.changePassword",
  "params": {
    "oldPassword": "OLD PASSWORD HERE",
    "newPassword": "NEW PASSWORD HERE"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/auth
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {},
  "id": 1
}
```

description: More information available regarding Avalanche Go APIs and learning how to interact with the C-Chain. **sidebar_position:** 4

Contract Chain (C-Chain) API

:::info Ethereum has its own notion of `networkID` and `chainID`. These have no relationship to Avalanche's view of networkID and chainID and are purely internal to the [C-Chain](#). On Mainnet, the C-Chain uses `1` and `43114` for these values. On the Fuji Testnet, it uses `1` and `43113` for these values.
`networkID` and `chainID` can also be obtained using the `net_version` and `eth_chainId` methods.

:::

Deploying a Smart Contract

[Deploy a Smart Contract on Avalanche Using Remix and MetaMask](#)

Ethereum APIs

Endpoints

JSON-RPC Endpoints

To interact with C-Chain via the JSON-RPC endpoint:

```
/ext/bc/C/rpc
```

To interact with other instances of the EVM via the JSON-RPC endpoint:

```
/ext/bc/blockchainID/rpc
```

where `blockchainID` is the ID of the blockchain running the EVM.

WebSocket Endpoints

:::info On the [public API node](#), it only supports C-Chain websocket API calls for API methods that don't exist on the C-Chain's HTTP API :::

To interact with C-Chain via the websocket endpoint:

```
/ext/bc/C/ws
```

For example, to interact with the C-Chain's Ethereum APIs via websocket on localhost you can use:

```
ws://127.0.0.1:9650/ext/bc/C/ws
```

:::tip

On localhost, use `ws://`. When using the [Public API](#) or another host that supports encryption, use `wss://`. :::

To interact with other instances of the EVM via the websocket endpoint:

```
/ext/bc/blockchainID/ws
```

where `blockchainID` is the ID of the blockchain running the EVM.

Standard Ethereum APIs

Avalanche offers an API interface identical to Geth's API except that it only supports the following services:

- `web3_`
- `net_`
- `eth_`
- `personal_`
- `txpool_`
- `debug_` (note: this is turned off on the public API node.)

You can interact with these services the same exact way you'd interact with Geth. See the [Ethereum Wiki's JSON-RPC Documentation](#) and [Geth's JSON-RPC Documentation](#) for a full description of this API.

:::info

For batched requests on the [public API node](#), the maximum number of items is 40. We are working on to support a larger batch size.

:::

Avalanche - Ethereum APIs

In addition to the standard Ethereum APIs, Avalanche offers `eth_getAssetBalance`, `eth_baseFee`, `eth_maxPriorityFeePerGas`, and `eth_getChainConfig`.

They use the same endpoint as standard Ethereum APIs:

```
/ext/bc/C/rpc
```

`eth_getAssetBalance`

Retrieves the balance of first class Avalanche Native Tokens on the C-Chain (excluding AVAX, which must be fetched with `eth_getBalance`).

:::note

The AssetID for AVAX differs depending on the network you are on.

Mainnet: FvwEAhmxFfeiG8SnEvq42hc6whRyY3EFYAvabMqDNDGCgxN5Z

Testnet: U8iRqJoiJm8xZHAacmvYyZVwqQx6uDNTQeP3CQ6fcgQk3JqnK

For finding the `assetID` of other assets, please note that `avax.getUTXOs` and `avax.getAtomicTx` return the `assetID` in their output.

...

Signature:

```
eth_getAssetBalance({  
    address: string,  
    blk: BlkNrOrHash,  
    assetID: string,  
}) -> {balance: int}
```

- `address` owner of the asset
- `blk` is the block number or hash at which to retrieve the balance
- `assetID` id of the asset for which the balance is requested

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "method": "eth_getAssetBalance",  
    "params": [  
        "0x8723e5773847A4Eb5FeEDabD9320802c5c812F46",  
        "latest",  
        "3RvKBAmQnfYionFXMfW5P8TDZgZiogKbHjM8cjpui6LKA9F5T"  
    ],  
    "id": 1  
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/rpc
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": "0x1388"  
}
```

eth_baseFee

Get the base fee for the next block.

Signature:

```
eth_baseFee() -> {}
```

`result` is the hex value of the base fee for the next block.

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "method": "eth_baseFee",  
    "params": []  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/rpc
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": "0x34630b8a00"  
}
```

eth_maxPriorityFeePerGas

Get the priority fee needed to be included in a block.

Signature:

```
eth_maxPriorityFeePerGas() -> {}
```

`result` is hex value of the priority fee needed to be included in a block.

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "method": "eth_maxPriorityFeePerGas",  
    "params": []  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/rpc
```

```
"id"      :1,
"method"  :"eth_maxPriorityFeePerGas",
"params"  :[]
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/rpc
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0x2540be400"
}
```

For more information on dynamic fees see the [C-Chain section of the transaction fee documentation](#).

eth_getChainConfig

eth_getChainConfig returns chain config. This API is enabled by default with `internal-eth` namespace.

Signature:

```
eth_getChainConfig({}) -> {chainConfig: json}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id"      :1,
  "method"  :"eth_getChainConfig",
  "params"  :[]
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/rpc
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "chainId": 43112,
    "homesteadBlock": 0,
    "daoForkBlock": 0,
    "daoForkSupport": true,
    "eip150Block": 0,
    "eip150Hash": "0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "istanbulBlock": 0,
    "muirGlacierBlock": 0,
    "apricotPhase1BlockTimestamp": 0,
    "apricotPhase2BlockTimestamp": 0,
    "apricotPhase3BlockTimestamp": 0,
    "apricotPhase4BlockTimestamp": 0,
    "apricotPhase5BlockTimestamp": 0
  }
}
```

Avalanche Specific APIs

To interact with the `avax` specific RPC calls on the C-Chain:

```
/ext/bc/C/avax
```

To interact with other instances of the EVM AVAX endpoints:

```
/ext/bc/blockchainID/avax
```

avax.export

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Export an asset from the C-Chain to X-Chain or P-Chain. After calling this method, you must call the X-Chain's [avm.import](#), or P-Chain's [platform.import](#).

Signature:

```
avax.export({
  to: string,
  amount: int,
  assetID: string,
  baseFee: int,
  username: string,
  password:string,
}) -> {txID: string}
```

- `to` is the X-Chain or P-Chain address the asset is sent to.
- `amount` is the amount of the asset to send.
- `assetID` is the ID of the asset. To export AVAX use "AVAX" as the `assetID`.
- `baseFee` is the base fee that should be used when creating the transaction. If omitted, a suggested fee will be used.
- `username` is the user that controls the address that transaction will be sent from.
- `password` is `username`'s password.

Example Call:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "id"      :1,
  "method"  :"avax.export",
  "params"  :{
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "amount": 500,
    "assetID": "2nzgmhZLuVq8jc7NNu2eahkKwoJcbFWXWJCxHBVVAJEZkhquoK",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "2W5JuFENitzKTpJsy9igBpTcEeBKxBHHGAUkgsSUnkjVVGQ9i8"
  },
  "id": 1
}
```

avax.exportAVAX

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

DEPRECATED—instead use [avax.export](#).

Send AVAX from the C-Chain to X-Chain or P-Chain. After calling this method, you must call the X-Chain's [avm.import](#) or P-Chain's [platform.import](#) with assetID `AVAX` on the X-Chain to complete the transfer.

Signature:

```
avax.exportAVAX({
  to: string,
  amount: int,
  baseFee: int,
  username: string,
  password:string,
}) -> {txID: string}
```

Request:

- `to` is X-Chain or P-Chain address the asset is sent to.
- `amount` is the amount of the asset to send.
- `baseFee` is the base fee that should be used when creating the transaction. If omitted, a suggested fee will be used.
- `username` is the user that controls the address that transaction will be sent from.
- `password` is `username`'s password.

Response:

- `txID` is the TXID of the completed ExportTx.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" :"avax.exportAVAX",
  "params" :{
    "from": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"],
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "amount": 500,
    "changeAddr": "0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "2ffcxdkikXXA4JdyRoS38dd7zoThkapNPeZuGPmmLBbiuBBHDa"
  },
  "id": 1
}
```

avax.exportKey

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Get the private key that controls a given address. The returned private key can be added to a user with `avax.importKey`.

Signature:

```
avax.exportKey({
  username: string,
  password:string,
  address:string
}) -> {privateKey: string}
```

Request:

- `username` must control `address`.
- `address` is the address for which you want to export the corresponding private key. It should be in hex format.

Response:

- `privateKey` is the CB58 encoded string representation of the private key that controls `address`. It has a `PrivateKey-` prefix and can be used to import a key via `avax.importKey`.
- `privateKeyHex` is the hex string representation of the private key that controls `address`. It can be used to import an account into MetaMask.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" :"avax.exportKey",
  "params" :{
    "username" :"myUsername",
    "password": "myPassword",
    "address": "0xc876DF0F099b3eb32cBB78820d39F5813f73E18C"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "privateKey": "PrivateKey-2o2uPgTSf3aR5nW6yLHjBEAiatAFKEhApvYzsjvAJKRXVWCYkE",
    "privateKeyHex": "0xec381fb8d32168be4cf7f8d4ce9d8ca892d77ba574264f3665ad5edb89710157"
  },
  "id": 1
}
```

avax.getAtomicTx

Gets a transaction by its ID. Optional encoding parameter to specify the format for the returned transaction. Can only be `hex` when a value is provided.

Signature:

```
avax.getAtomicTx({
  txID: string,
  encoding: string, //optional
}) -> {
  tx: string,
  encoding: string,
  blockHeight: string
}
}
```

Request:

- `txID` is the transaction ID. It should be in cb58 format.
 - `encoding` is the encoding format to use. Can only be `hex` when a value is provided.

Response:

- `tx` is the transaction encoded to `encoding`.
 - `encoding` is the `encoding`.
 - `blockHeight` is the height of the block which the transaction was included in.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "avax.getAtomicTx",
    "params": {
        "txID": "2GD5SRYJQr2kw5jE73trBfIAgVQyrCaeg223TaTyJFYXf2kPty",
        "encoding": "hex"
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "tx": "0x0000000000000000000000000000000030399d0775f450604bd2fbca9ce0c5c1c6df2dc2acb8c92c26eeae6e6df4502b19d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3c  
        "encoding": "hex",  
        "blockHeight": "1"  
    },  
    "id": 1  
}
```

`avax.getAtomicTxStatus`

Get the status of an atomic transaction sent to the network.

Signature:

```
avax.getAtomicTxStatus((txID: string)) -> {
  status: string,
  blockHeight: string // returned when status is Accepted
}
```

status is one of:

- Accepted : The transaction is (or will be) accepted by every node. Check the `blockHeight` property
 - Processing : The transaction is being voted on by this node
 - Dropped : The transaction was dropped by this node because it thought the transaction invalid
 - Unknown : The transaction hasn't been seen by this node

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id"      : 1,
    "method"  : "avax.getAtomicTxStatus",
    "params"  : {
        "txID": "2QouvFWUbjuySRxeX5xMbNCuAaKWFbk5FeEa2JmoF85RKLk2dD"
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "status": "Accepted",
    "blockHeight": "1"
  }
}
```

avax.getUTXOs

Gets the UTXOs that reference a given address.

Signature:

```
avax.getUTXOs(
  {
    addresses: string,
    limit: int, //optional
    startIndex: { //optional
      address: string,
      utxo: string
    },
    sourceChain: string,
    encoding: string, //optional
  },
) ->
{
  numFetched: int,
  utxos: []string,
  endIndex: {
    address: string,
    utxo: string
  }
}
```

- `utxos` is a list of UTXOs such that each UTXO references at least one address in `addresses`.
- At most `limit` UTXOs are returned. If `limit` is omitted or greater than 1024, it is set to 1024.
- This method supports pagination. `endIndex` denotes the last UTXO returned. To get the next set of UTXOs, use the value of `endIndex` as `startIndex` in the next call.
- If `startIndex` is omitted, will fetch all UTXOs up to `limit`.
- When using pagination (that is when `startIndex` is provided), UTXOs are not guaranteed to be unique across multiple calls. That is, a UTXO may appear in the result of the first call, and then again in the second call.
- When using pagination, consistency is not guaranteed across multiple calls. That is, the UTXO set of the addresses may have changed between calls.
- `encoding` sets the format for the returned UTXOs. Can only be `hex` when a value is provided.

Example

Suppose we want all UTXOs that reference at least one of `C-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5`.

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avax.getUTXOs",
  "params": {
    "addresses": ["C-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "sourceChain": "X",
    "startIndex": {
      "address": "C-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "22RXW7SWjBrrxu2vzDkd8uza7fuEmNpgbj58CxBob9UbP37HSB"
    },
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

This gives response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "3",
    "utxos": [
      "0x0000a799e7448acf74ca9223159a04f93b948f99cf28509f908839532b2f85baafffc300000001dbc890f77f49b96857648b72b77f9f82937f28a68704af05da",
      "0x00006385c683d43bdbe754c224be36c5004ea7ce49c0849cadeaea6af93dae18cc7700000001dbc890f77f49b96857648b72b77f9f82937f28a68704af05da",
      "0x00006385c683d43bdbe754c224be36c5004ea7ce49c0849cadeaea6af93dae18cc7700000001dbc890f77f49b96857648b72b77f9f82937f28a68704af05da"
    ]
  }
}
```

```

    "0x000038137283c94582351b86c3e90808312636769e3f5c14fbf1152d6634f770695c0000001dbc890f77f49b96857648b72b77f9f82937f28a68704af05da(
    ],
    "endIndex": {
      "address": "C-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "0x9333ef8a05f26acf2d8766f94723f749870fa2ca80c19c33cc945d79013d7c50fd023beb"
    },
    "encoding": "hex"
  },
  "id": 1
}

```

`avax.import`

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Finalize the transfer of a non-AVAX or AVAX from X-Chain or P-Chain to the C-Chain. Before this method is called, you must call the X-Chain's [avm.export](#) or P-Chain's [platform.exportAVAX](#) with assetID `AVAX` to initiate the transfer.

Signature:

```

avax.import({
  to: string,
  sourceChain: string,
  baseFee: int, // optional
  username: string,
  password:string,
}) -> {txID: string}

```

Request:

- `to` is the address the asset is sent to. This must be the same as the `to` argument in the corresponding call to the X-Chain's or P-Chain's `export`.
- `sourceChain` is the ID or alias of the chain the asset is being imported from. To import funds from the X-Chain, use `"X"`; for the P-Chain, use `"P"`.
- `baseFee` is the base fee that should be used when creating the transaction. If omitted, a suggested fee will be used.
- `username` is the user that controls the address that transaction will be sent from.
- `password` is `username`'s password.

Response:

- `txID` is the ID of the completed ImportTx.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avax.import",
  "params": {
    "to": "0xb879aff6b3d24352Ac1985c1F45BA4c3493A398",
    "sourceChain": "X",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax

```

Example Response:

```

{
  "jsonrpc": "2.0",
  "result": {
    "txID": "6bJq9dbghiQvoshT3uSubg9oB24n7Ei6MLnxvrdmao78oHR9t"
  },
  "id": 1
}

```

`avax.importAVAX`

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

DEPRECATED—instead use [avax.import](#)

Finalize a transfer of AVAX from the X-Chain or P-Chain to the C-Chain. Before this method is called, you must call the X-Chain's [avm.export](#) or P-Chain's [platform.exportAVAX](#) with assetID `AVAX` to initiate the transfer.

Signature:

```

avax.importAVAX({
  to: string,
}

```

```
sourceChain: string,
baseFee: int, // optional
username: string,
password:string,
}) -> {txID: string}
```

Request:

- `to` is the address the AVAX is sent to. It should be in hex format.
- `sourceChain` is the ID or alias of the chain the AVAX is being imported from. To import funds from the X-Chain, use `"x"` ; for the P-Chain, use `"p"` .
- `baseFee` is the base fee that should be used when creating the transaction. If omitted, a suggested fee will be used.
- `username` is the user that controls the address that transaction will be sent from.
- `password` is `username`'s password.

Response:

- `txID` is the ID of the completed ImportTx.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avax.importAVAX",
  "params": {
    "to": "0x4b879aff6b3d24352Ac1985c1F45BA4c3493A398",
    "sourceChain": "X",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "LWTRsiKnEUJC58y8ezAk6hhzmSMUCtemLvm3LZFw8fxDQpns3"
  },
  "id": 1
}
```

avax.importKey

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Give a user control over an address by providing the private key that controls the address.

Signature:

```
avax.importKey({
  username: string,
  password:string,
  privateKey:string
}) -> {address: string}
```

Request:

- Add `privateKey` to `username`'s set of private keys.

Response:

- `address` is the address `username` now controls with the private key. It will be in hex format.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avax.importKey",
  "params": {
    "username": "myUsername",
    "password": "myPassword",
    "privateKey": "PrivateKey-2o2uPgTSf3aR5nW6yLHjBEAiataFKEhApvYzsjvAJKRXVWCYkE"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/avax
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "address": "0xc876DF0F099b3eb32cBB78820d39F5813f73E18C"  
  },  
  "id": 1  
}
```

avax.issueTx

Send a signed transaction to the network. `encoding` specifies the format of the signed transaction. Can only be `hex` when a value is provided.

Signature:

```
avax.issueTx({
    tx: string,
    encoding: string, //optional
}) -> {
    txID: string
}
```

Example Call:

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "txID": "NUPlwbt2hsYxpQg4H2o451hmTWQ4JZx2zMzM4SinwtHgAdX1JLPHXvWSXEnepecStLj"  
    }  
}
```

Admin API

This API can be used for debugging. Note that the Admin API is disabled by default. To run a node with the Admin API enabled, use [C-Chain config flag --coreth-admin-api-enabled:true](#) .

Endpoint

/ext/bc/C/admin

On the last word of the 9th line

```
admin.setLanguage((languageString))
```

- Log Level** is the log level to be set.

Example Cells

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.setLogLevel",
  "params": {
    "level": "INFO"
  }
}' http://127.0.0.1:8080/jsonrpc
```

```
        }
    }' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
```

admin.startCPUProfiler

Starts a CPU profile.

Signature:

```
admin.startCPUProfiler() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.startCPUProfiler",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
```

admin.stopCPUProfiler

Stops and writes a CPU profile.

Signature:

```
admin.stopCPUProfiler() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.stopCPUProfiler",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
```

admin.memoryProfile

Runs and writes a memory profile.

Signature:

```
admin.memoryProfile() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
```

```
"method" :"admin.memoryProfile",
"params": {}
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

admin.lockProfile

Runs a mutex profile writing to the `coreth_performance_c` directory.

Signature:

```
admin.lockProfile() -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" :1,
  "method" :"admin.lockProfile",
  "params": {}
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/C/admin
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

sidebar_position: 8 **description:** This API can be used for measuring node health. Use Avalanche docs as a reference when testing node health. Helpful examples provided.

Health API

This API can be used for measuring node health.

:::info

This API set is for a specific node, it is unavailable on the [public server](#).

:::

Filterable Health Checks

The health checks that are run by the node are filterable. You can specify which health checks you want to see by using `tags` filters. Returned results will only include health checks that match the specified tags and global health checks like `network`, `database` etc. When filtered, the returned results will not show the full node health, but only a subset of filtered health checks. This means the node can be still unhealthy in unfiltered checks, even if the returned results show that the node is healthy. AvalancheGo supports filtering tags by subnetIDs. For more information check Filtering sections below.

GET Request

To get an HTTP status code response that indicates the node's health, make a `GET` request to `/ext/health`. If the node is healthy, it will return a `200` status code. If you want more in-depth information about a node's health, use the JSON RPC methods.

Filtering

To filter GET health checks, add a `tag` query parameter to the request. The `tag` parameter is a string. To filter health results by subnetID, use the `subnetID` tag. For example, to filter health results by subnetID `29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL`, use the following query:

```
curl --location --request GET 'http://localhost:9650/ext/health?tag=29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL' \
--header 'Content-Type: application/json' \
--data=raw '{
  "jsonrpc": "2.0",
  "id" :1,
```

```
"method" :"health.health",
}'
```

In this example returned results will contain global health checks and health checks that are related to subnetID
29uVeLPJBleQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL .

Note: This filtering can show healthy results even if the node is unhealthy in other Chains/Subnets.

In order to filter results by multiple tags, use multiple `tag` query parameters. For example, to filter health results by subnetID

29uVeLPJBleQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL and 28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY use the following query:

```
curl --location --request GET 'http://localhost:9650/ext/health?
tag=29uVeLPJBleQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL&tag=28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc":"2.0",
    "id"      :1,
    "method"  :"health.health",
}'
```

Returned results will contain checks for both subnetIDs and global health checks.

JSON RPC Request

Format

This API uses the `json 2.0` RPC format. For more information on making JSON RPC calls, see [here](#).

Endpoint

```
/ext/health
```

Methods

`health.health`

The node runs a set of health checks every 30 seconds, including a health check for each chain. This method returns the last set of health check results.

Signature:

```
health.health() -> {
    checks: []{
        checkName: {
            message: JSON,
            error: JSON,
            timestamp: string,
            duration: int,
            contiguousFailures: int,
            timeOfFirstFailure: int
        }
    },
    healthy: bool
}
```

`healthy` is true if the node if all health checks are passing.

`checks` is a list of health check responses.

- A check response may include a `message` with additional context.
- A check response may include an `error` describing why the check failed.
- `timestamp` is the timestamp of the last health check.
- `duration` is the execution duration of the last health check, in nanoseconds.
- `contiguousFailures` is the number of times in a row this check failed.
- `timeOfFirstFailure` is the time this check first failed.

More information on these measurements can be found in the documentation for the [go-sundheit](#) library.

Example Call:

```
curl -X POST --data '{
    "jsonrpc":"2.0",
    "id"      :1,
    "method"  :"health.health"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/health
```

Example Response:

In this example response, the C-Chain's health check is failing.

```
{
  "jsonrpc": "2.0",
  "result": {
    "checks": {
      "C": {
        "message": null,
        "error": {
          "message": "example error message"
        },
        "timestamp": "2020-10-14T14:04:20.577596622",
        "duration": 465253,
        "contiguousFailures": 50,
        "timeOfFirstFailure": "2020-10-14T13:16:10.576435413Z"
      },
      "P": {
        "message": {
          "percentConnected": 0.9967694992864075
        },
        "timestamp": "2020-10-14T14:04:08.668743851Z",
        "duration": 433363830,
        "contiguousFailures": 0,
        "timeOfFirstFailure": null
      },
      "X": {
        "timestamp": "2020-10-14T14:04:20.3962705Z",
        "duration": 1853,
        "contiguousFailures": 0,
        "timeOfFirstFailure": null
      }
    },
    "chains.default.bootstrapped": {
      "timestamp": "2020-10-14T14:04:04.238623814Z",
      "duration": 8075,
      "contiguousFailures": 0,
      "timeOfFirstFailure": null
    },
    "network.validators.heartbeat": {
      "message": {
        "heartbeat": 1602684245
      },
      "timestamp": "2020-10-14T14:04:05.610007874Z",
      "duration": 6124,
      "contiguousFailures": 0,
      "timeOfFirstFailure": null
    }
  },
  "healthy": false
},
"id": 1
}
```

Filtering

JSON RPC methods in Health API supports filtering by tags. In order to filter results use `tags` params in the request body. `tags` accepts a list of tags.

Currently only `subnetID`s are supported as tags. For example, to filter health results by `subnetID`

`29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL` use the following request:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "health.health",
  "params": {
    "tags": ["29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL"]
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/health
```

Returned results will contain checks for subnetID

`29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL` and global health checks.

`sidebar_position: 9`

Index API

AvalancheGo can be configured to run with an indexer. That is, it saves (indexes) every container (a block, vertex or transaction) it accepts on the X-Chain, P-Chain and C-Chain. To run AvalancheGo with indexing enabled, set command line flag `--index-enabled` to true. **AvalancheGo will only index containers that are accepted when running with --index-enabled set to true.** To ensure your node has a complete index, run a node with a fresh database and `--index-enabled` set to true. The node will accept every block, vertex and transaction in the network history during bootstrapping, ensuring your index is complete. It is OK to turn off your node if it is running with indexing enabled. If it restarts with indexing still enabled, it will accept all containers that were accepted while it was offline. The indexer should never fail to index an accepted block, vertex or transaction.

Indexed containers (that is, accepted blocks, vertices and transactions) are timestamped with the time at which the node accepted that container. Note that if the container was indexed during bootstrapping, other nodes may have accepted the container much earlier. Every container indexed during bootstrapping will be timestamped with the time at which the node bootstrapped, not when it was first accepted by the network.

Note that for DAGs (including the X-Chain), nodes may accept vertices and transactions in a different order from one another.

If `--index-enabled` is changed to `false` from `true`, AvalancheGo won't start as doing so would cause a previously complete index to become incomplete, unless the user explicitly says to do so with `--index-allow-incomplete`. This protects you from accidentally running with indexing disabled, after previously running with it enabled, which would result in an incomplete index.

This document shows how to query data from AvalancheGo's Index API. The Index API is only available when running with `--index-enabled`.

Go Client

There is a Go implementation of an Index API client. See documentation [here](#). This client can be used inside a Go program to connect to an AvalancheGo node that is running with the Index API enabled and make calls to the Index API.

Format

This API uses the `json 2.0` RPC format. For more information on making JSON RPC calls, see [here](#).

Endpoints

Each chain has one or more index. To see if a C-Chain block is accepted, for example, send an API call to the C-Chain block index. To see if an X-Chain vertex is accepted, for example, send an API call to the X-Chain vertex index.

C-Chain Blocks

```
/ext/index/C/block
```

P-Chain Blocks

```
/ext/index/P/block
```

X-Chain Transactions

```
/ext/index/x/tx
```

X-Chain Blocks

```
/ext/index/X/block
```

:::caution

To ensure historical data can be accessed, the `/ext/index/X/vtx` is still accessible, even though it is no longer populated with chain data since the Cortina activation. If you are using `V1.10.0` or higher, you need to migrate to using the `/ext/index/X/block` endpoint.

:::

Methods

`index.getContainerByID`

Get container by ID.

Signature:

```
index.getContainerByID({
  id: string,
  encoding: string
}) -> {
  id: string,
  bytes: string,
  timestamp: string,
  encoding: string,
  index: string
}
```

Request:

- `id` is the container's ID
- `encoding` is "hex" only.

Response:

- `id` is the container's ID
- `bytes` is the byte representation of the container
- `timestamp` is the time at which this node accepted the container
- `encoding` is "hex" only.
- `index` is how many containers were accepted in this index before this one

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx' \
--header 'Content-Type: application/json' \
--data=raw '{
    "jsonrpc": "2.0",
    "method": "index.getContainerByID",
    "params": {
        "id": "6fxF5hncR8LXvwtM8iezFQBpK5cubV6y1dWgpJCCNyzGB1EzY",
        "encoding": "hex"
    },
    "id": 1
}'
```

Example Response:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "id": "6fxF5hncR8LXvwtM8iezFQBpK5cubV6y1dWgpJCCNyzGB1EzY",
        "bytes": "0x000000000040003039d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf00000001dbcfc890f77f49b96857648b72b77f9f8293",
        "timestamp": "2021-04-02T15:34:00.262979-07:00",
        "encoding": "hex",
        "index": "0"
    }
}
```

index.getContainerByIndex

Get container by index. The first container accepted is at index 0, the second is at index 1, etc.

Signature:

```
index.getContainerByIndex({
    index: uint64,
    encoding: string
}) -> {
    id: string,
    bytes: string,
    timestamp: string,
    encoding: string,
    index: string
}
```

Request:

- `index` is how many containers were accepted in this index before this one
- `encoding` is "hex" only.

Response:

- `id` is the container's ID
- `bytes` is the byte representation of the container
- `timestamp` is the time at which this node accepted the container
- `index` is how many containers were accepted in this index before this one
- `encoding` is "hex" only.

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx' \
--header 'Content-Type: application/json' \
--data=raw '{
    "jsonrpc": "2.0",
```

```
"method": "index.getContainerByIndex",
"params": {
    "index": 0,
    "encoding": "hex"
},
"id": 1
}'
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "id": "6fxF5hncR8LxvwtM8iezFQBpK5cubV6y1dWgpJCeNyzGB1EzY",  
        "bytes":  
"0x000000000000400003039d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf00000001dbcfa890f77f49b96857648b72b77f9f8293"  
        "timestamp": "2021-04-02T15:34:00.262979-07:00",  
        "encoding": "hex",  
        "index": "0"  
    }  
}
```

```
index.getContainerRange
```

Returns the transactions at index [startIndex], [startIndex+1], ..., [startIndex+n-1]

- If `[n] == 0`, returns an empty response (for example: `null`).
 - If `[startIndex] >` the last accepted index, returns an error (unless the above apply).
 - If `[n] > [MaxFetchedByRange]`, returns an error.
 - If we run out of transactions, returns the ones fetched before running out.
 - `numOfFetch` must be in `[0, 1024]`.

Signature:

```
index.getContainerRange((  
    startIndex: uint64,  
    numToFetch: uint64,  
    encoding: string  
) => []{  
    id: string,  
    bytes: string,  
    timestamp: string,  
    encoding: string,  
    index: string  
})
```

Request:

- `startIndex` is the beginning index
 - `numToFetch` is the number of containers to fetch
 - `encoding` is "hex" only.

Response:

- `id` is the container's ID
 - `bytes` is the byte representation of the container
 - `timestamp` is the time at which this node accepted the container
 - `encoding` is "hex" only.
 - `index` is how many containers were accepted in this index before this one

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx'  
--header 'Content-Type: application/json' \  
--data-raw '{  
    "jsonrpc": "2.0",  
    "method": "index.getContainerRange",  
    "params": {  
        "startIndex":0,  
        "numToFetch":100,  
        "encoding": "hex"  
    },  
    "id": 1  
}'
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "id": "6fXf5hncR8LxvwtM8iezFQBpK5cubV6yldWgpJCCNyzGB1EzY",
      "bytes": "0x0000000000400003039d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6f00000001dbc890f77f49b96857648b72b77f9f8293",
      "timestamp": "2021-04-02T15:34:00.262979-07:00",
      "encoding": "hex",
      "index": "0"
    }
  ]
}
```

index.getIndex

Get a container's index.

Signature:

```
index.getIndex({
  id: string,
  encoding: string
}) -> {
  index: string
}
```

Request:

- `id` is the ID of the container to fetch
- `encoding` is "hex" only.

Response:

- `index` is how many containers were accepted in this index before this one

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx' \
--header 'Content-Type: application/json' \
--data=raw '{
  "jsonrpc": "2.0",
  "method": "index.getIndex",
  "params": {
    "id": "6fXf5hncR8LxvwtM8iezFQBpK5cubV6yldWgpJCCNyzGB1EzY",
    "encoding": "hex"
  },
  "id": 1
}'
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "index": "0"
  },
  "id": 1
}
```

index.getLastAccepted

Get the most recently accepted container.

Signature:

```
index.getLastAccepted({
  encoding: string
}) -> {
  id: string,
  bytes: string,
  timestamp: string,
  encoding: string,
  index: string
}
```

Request:

- `encoding` is "hex" only.

Response:

- `id` is the container's ID
- `bytes` is the byte representation of the container
- `timestamp` is the time at which this node accepted the container
- `encoding` is "hex" only.

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "method": "index.getLastAccepted",
    "params": {
        "encoding": "hex"
    },
    "id": 1
}'
```

Example Response:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "id": "6fxf5hncR8LxvwtM8iezFQBpK5cubV6y1dWgpJCCNyzGB1EzY",
        "bytes": "0x0000000000400003039d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf00000001dbcfc890f77f49b96857648b72b77f9f8293"
        "timestamp": "2021-04-02T15:34:00.262979-07:00",
        "encoding": "hex",
        "index": "0"
    }
}
```

index.isAccepted

Returns true if the container is in this index.

Signature:

```
index.isAccepted({
  id: string,
  encoding: string
}) -> {
  isAccepted: bool
}
```

Request:

- `id` is the ID of the container to fetch
- `encoding` is "hex" only.

Response:

- `isAccepted` displays if the container has been accepted

Example Call:

```
curl --location --request POST 'localhost:9650/ext/index/X/tx' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc": "2.0",
    "method": "index.isAccepted",
    "params": {
        "id": "6fxf5hncR8LxvwtM8iezFQBpK5cubV6y1dWgpJCCNyzGB1EzY",
        "encoding": "hex"
    },
    "id": 1
}'
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "isAccepted": true
  },
  "id": 1
}
```

Example: Iterating Through X-Chain Transaction

Here is an example of how to iterate through all transactions on the X-Chain.

:::warning To help users to try out this example and other index APIs, we have set up a testing indexer node located at <https://indexer-demo.avax.network>. This indexer node is not for production use. We may change or shut it down at any time without notice. :::

You can use the Index API to get the ID of every transaction that has been accepted on the X-Chain, and use the X-Chain API method `avm.getTx` to get a human-readable representation of the transaction.

To get an X-Chain transaction by its index (the order it was accepted in), use Index API method [index.getLastAccepted](#).

For example, to get the second transaction (note that `"index":1`) accepted on the X-Chain, do:

```
curl --location --request POST 'https://indexer-demo.avax.network/ext/index/x/tx' \
--header 'Content-Type: application/json' \
--data-raw '{
  "jsonrpc": "2.0",
  "method": "index.getContainerByIndex",
  "params": {
    "encoding": "hex",
    "index": 1
  },
  "id": 1
}'
```

This returns the ID of the second transaction accepted in the X-Chain's history. To get the third transaction on the X-Chain, use `"index":2`, and so on.

The above API call gives the response below:

```
{
  "jsonrpc": "2.0",
  "result": {
    "id": "ZGYTSU8w3zUP6VFseGC798vA2Vnxnfj6fz1QPfA9N93bhjJvo",
    "bytes": "0x000000000000000000000000000000001ed5f38341e436e5d46e2bb00b45d62ae97d1b050c64bc634ae10626739e35c4b0000000221e67317cbc4be2aeb00677ad6462778a8f",
    "timestamp": "2021-11-04T00:42:55.01643414Z",
    "encoding": "hex",
    "index": "1"
  },
  "id": 1
}
```

The ID of this transaction is `ZGYTSU8w3zUP6VFseGC798vA2Vnxnfj6fz1QPfA9N93bhjJvo`.

To get the transaction by its ID, use API method `avm.getTx`:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getTx",
  "params": {
    "txID": "ZGYTSU8w3zUP6VFseGC798vA2Vnxnfj6fz1QPfA9N93bhjJvo",
    "encoding": "json"
  }
}' -H 'content-type:application/json;' https://api.avax.network/ext/bc/x
```

Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "tx": {
      "unsignedTx": {
        "networkID": 1,
        "blockchainID": "2oYMBNV4eNHqk2fjjv5nQLDtmNJzq5s3qs3Lo6ftnC6FBM",
        "outputs": [
          {
            "id": 1
          }
        ]
      }
    }
  }
}
```

```

    "assetID": "FvwEAhmxAfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
    "fxID": "spdxUxVJQbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEP3h6TLuxqQ",
    "output": {
      "addresses": ["X-avax1wst8jt3z3fm9ce0z6akj3266zmgccdp03hj1aj"],
      "amount": 4999000000,
      "locktime": 0,
      "threshold": 1
    }
  },
  {
    "assetID": "FvwEAhmxAfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
    "fxID": "spdxUxVJQbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEP3h6TLuxqQ",
    "output": {
      "addresses": ["X-avax1s1t2dhfu6a6qezcn5sgtagumq8ag8we75f84sw"],
      "amount": 2347999000000,
      "locktime": 0,
      "threshold": 1
    }
  }
],
"inputs": [
  {
    "txID": "qysTYUMCWdsR3MctzyfXiSvoSf6evbeFGRLLzA4j2BjNXTknh",
    "outputIndex": 0,
    "assetID": "FvwEAhmxAfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
    "fxID": "spdxUxVJQbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEP3h6TLuxqQ",
    "input": {
      "amount": 2352999000000,
      "signatureIndices": [0]
    }
  }
],
"memo": "0x"
},
"credentials": [
  {
    "fxID": "spdxUxVJQbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEP3h6TLuxqQ",
    "credential": {
      "signatures": [
        "0xeb83d3d29f1247efb4a3a1141ab5c966f46f946f9c943b9bc19f858bd416d10060c23d5d9c7db3a0da23446b97cd9cf9f8e61df98e1b1692d764c84a686f5f8
      ]
    }
  }
],
"encoding": "json"
},
"id": 1
}

```

sidebar_position: 10

Info API

This API can be used to access basic information about the node.

:::info This API set is for a specific node, it is unavailable on the [public server](#). :::

Format

This API uses the `json 2.0` RPC format. For more information on making JSON RPC calls, see [here](#).

Endpoint

```
/ext/info
```

Methods

`info.isBootstrapped`

Check whether a given chain is done bootstrapping

Signature:

```
info.isBootstrapped({chain: string}) -> {isBootstrapped: bool}
```

chain is the ID or alias of a chain.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.isBootstrapped",
  "params": {
    "chain": "X"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "isBootstrapped": true
  },
  "id": 1
}
```

info.getBlockchainID

Given a blockchain's alias, get its ID. (See [admin.aliasChain](#).)

Signature:

```
info.getBlockchainID({alias:string}) -> {blockchainID:string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getBlockchainID",
  "params": {
    "alias": "X"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "blockchainID": "sV6o671RtkGBcnolFiaDbVcFv2sG5aVXMZYzKdP4VQAWmJQnM"
  }
}
```

info.getNetworkID

Get the ID of the network this node is participating in.

Signature:

```
info.getNetworkID() -> {networkID:int}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNetworkID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
```

```
"id": 1,
"result": {
  "networkID": "2"
}
}
```

info.getNetworkName

Get the name of the network this node is participating in.

Signature:

```
info.getNetworkName() -> {networkName:string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNetworkName"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "networkName": "local"
  }
}
```

info.getNodeID

Get the ID of this node.

Signature:

```
info.getNodeID() -> {
  nodeID: string,
  nodePOP: {
    publicKey: string,
    proofOfPossession: string
  }
}
```

- `nodeID` is this node's ID
- `nodePOP` is this node's BLS key and proof of possession

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD",
    "nodePOP": {
      "publicKey": "0x8f95423f7142d00a48e1014a3de8d28907d420dc33b3052a6dee03a3f2941a393c2351e354704ca66a3fc29870282e15",
      "proofOfPossession": "0x86a3ab4c45cf31cae34c1d06f212434ac71b1be6cf0e46c80c162e057614a94a5bc9f1ded1a7029deb0ba4ca7c9b71411e293438691be79c2dbf19d1ca7c3ea"
    }
  },
  "id": 1
}
```

info.getNodeIP

Get the IP of this node.

Signature:

```
info.getNodeIP() -> {ip: string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeIP"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "ip": "192.168.1.1:9651"
  },
  "id": 1
}
```

info.getNodeVersion

Get the version of this node.

Signature:

```
info.getNodeVersion() -> {
  version: string,
  databaseVersion: string,
  gitCommit: string,
  vmVersions: map[string]string,
  rpcProtocolVersion: string,
}
```

where:

- `version` is this node's version
- `databaseVersion` is the version of the database this node is using
- `gitCommit` is the Git commit that this node was built from
- `vmVersions` is map where each key/value pair is the name of a VM, and the version of that VM this node runs
- `rpcProtocolVersion` is the RPCChainVM protocol version

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeVersion"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "version": "avalanche/1.9.1",
    "databaseVersion": "v1.4.5",
    "rpcProtocolVersion": "18",
    "gitCommit": "79cd09ba728e1cecef40acd60702f0a2d41ea404",
    "vmVersions": {
      "avm": "v1.9.1",
      "evm": "v0.11.1",
      "platform": "v1.9.1"
    },
    "id": 1
}
```

info.getTxFee

Get the fees of the network.

Signature:

```

info.getTxFee() ->
{
    txFee: uint64,
    createAssetTxFee: uint64,
    createSubnetTxFee: uint64,
    transformSubnetTxFee: uint64,
    createBlockchainTxFee: uint64,
    addPrimaryNetworkValidatorFee: uint64,
    addPrimaryNetworkDelegatorFee: uint64,
    addSubnetValidatorFee: uint64,
    addSubnetDelegatorFee: uint64
}

```

- `txFee` is the default fee for making transactions.
- `createAssetTxFee` is the fee for creating a new asset.
- `createSubnetTxFee` is the fee for creating a new Subnet.
- `transformSubnetTxFee` is the fee for converting a PoA Subnet into a PoS Subnet.
- `createBlockchainTxFee` is the fee for creating a new blockchain.
- `addPrimaryNetworkValidatorFee` is the fee for adding a new primary network validator.
- `addPrimaryNetworkDelegatorFee` is the fee for adding a new primary network delegator.
- `addSubnetValidatorFee` is the fee for adding a new Subnet validator.
- `addSubnetDelegatorFee` is the fee for adding a new Subnet delegator.

All fees are denominated in nAVAX.

Example Call:

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "info.getTxFee"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info

```

Example Response:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "txFee": "1000000",
        "createAssetTxFee": "1000000",
        "createSubnetTxFee": "1000000000",
        "transformSubnetTxFee": "1000000000",
        "createBlockchainTxFee": "1000000000",
        "addPrimaryNetworkValidatorFee": "0",
        "addPrimaryNetworkDelegatorFee": "0",
        "addSubnetValidatorFee": "1000000",
        "addSubnetDelegatorFee": "1000000"
    }
}
```

info.getVMs

Get the virtual machines installed on this node.

Signature:

```

info.getVMs() -> {
    vms: map[string][]string
}

```

Example Call:

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "info.getVMs",
    "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info

```

Example Response:

```
{
    "jsonrpc": "2.0",
    "result": {

```

```

    "vms": {
      "jyYyfQTxGMJLuGWa55kdP2p2zSUYsQ5Raupu4TW34ZAUBAbtq": ["avm"],
      "mgj786NP7uDwBCCq6YwThhaN8FLyybkCa4zBWTQbNgmK6k9A6": ["evm"],
      "qd2U4HDWUvMrVUEtcCHp6xH3Qpnn1XbU5MDdnBoiffFqvgXwT": ["nftfx"],
      "rWhpuQPF1kb72esV2nomhMuTYGkEb1oL29pt2EBXNmSy4kxn": ["platform"],
      "rXJsCSEYYXg2TehWxCCEEGj6JU2PWKTkd6cBdNLjoe2SpssKD9cy": ["propertyfx"],
      "spdxUxVJQbX85MGxMhbKw1sHxMnSqJ3QBzDyDYEPrh6TLuxqQ": ["secp256k1fx"]
    },
    "id": 1
  }
}

```

info.peers

Get a description of peer connections.

Signature:

```

info.peers({
  nodeIDs: string[] // optional
}) ->
{
  numPeers: int,
  peers: [] {
    ip: string,
    publicIP: string,
    nodeID: string,
    version: string,
    lastSent: string,
    lastReceived: string,
    benched: string[],
    observedUptime: int,
    observedSubnetUptime: map[string]int,
  }
}

```

- `nodeIDs` is an optional parameter to specify what NodeID's descriptions should be returned. If this parameter is left empty, descriptions for all active connections will be returned. If the node is not connected to a specified NodeID, it will be omitted from the response.
- `ip` is the remote IP of the peer.
- `publicIP` is the public IP of the peer.
- `nodeID` is the prefixed Node ID of the peer.
- `version` shows which version the peer runs on.
- `lastSent` is the timestamp of last message sent to the peer.
- `lastReceived` is the timestamp of last message received from the peer.
- `benched` shows chain IDs that the peer is being benched.
- `observedUptime` is this node's primary network uptime, observed by the peer.
- `observedSubnetUptime` is a map of Subnet IDs to this node's Subnet uptimes, observed by the peer.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.peers",
  "params": {
    "nodeIDs": []
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "numPeers": 3,
    "peers": [
      {
        "ip": "206.189.137.87:9651",
        "publicIP": "206.189.137.87:9651",
        "nodeID": "NodeID-8PYXX47kqLDe2wD4oPbvRRchcnSzMA4J4",
        "version": "avalanche/1.9.4",
        "lastSent": "2020-06-01T15:23:02Z",
        "lastReceived": "2020-06-01T15:22:57Z",
        "benched": [],
        "observedUptime": "99",
        "observedSubnetUptime": {}
      }
    ]
  }
}
```

```

        "observedSubnetUptimes": {},
        "trackedSubnets": [],
        "benched": []
    },
    {
        "ip": "158.255.67.151:9651",
        "publicIP": "158.255.67.151:9651",
        "nodeID": "NodeID-C14frln8EYNKyDfYixJ3rxSAVqTY3a8BB",
        "version": "avalanche/1.9.4",
        "lastSent": "2020-06-01T15:23:02Z",
        "lastReceived": "2020-06-01T15:22:34Z",
        "benched": [],
        "observedUptime": "75",
        "observedSubnetUptimes": {
            "29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL": "100"
        },
        "trackedSubnets": [
            "29uVeLPJB1eQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL"
        ],
        "benched": []
    },
    {
        "ip": "83.42.13.44:9651",
        "publicIP": "83.42.13.44:9651",
        "nodeID": "NodeID-LPbcSMGJ4yocxYxvS2kJ6umWeeFbctYZ",
        "version": "avalanche/1.9.3",
        "lastSent": "2020-06-01T15:23:02Z",
        "lastReceived": "2020-06-01T15:22:55Z",
        "benched": [],
        "observedUptime": "95",
        "observedSubnetUptimes": {},
        "trackedSubnets": [],
        "benched": []
    }
]
}
}

```

info.uptime

Returns the network's observed uptime of this node. This is the only reliable source of data for your node's uptime. Other sources may be using data gathered with incomplete (limited) information.

Signature:

```

info.uptime({
    subnetID: string // optional
}) ->
{
    rewardingStakePercentage: float64,
    weightedAveragePercentage: float64
}

```

- `subnetID` is the Subnet to get the uptime of. If not provided, returns the uptime of the node on the primary network.
- `rewardingStakePercentage` is the percent of stake which thinks this node is above the uptime requirement.
- `weightedAveragePercentage` is the stake-weighted average of all observed uptimes for this node.

Example Call:

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "info.uptime"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info

```

Example Response:

```

{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "rewardingStakePercentage": "100.0000",
        "weightedAveragePercentage": "99.0000"
    }
}

```

```
}
```

Example Subnet Call

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.uptime",
  "params": {
    "subnetID": "29uVeLPJBleQJkzRemU8g8wZDw5uJRqpab5U2mX9euieVwiEbL"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Example Subnet Response

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "rewardingStakePercentage": "74.0741",
    "weightedAveragePercentage": "72.4074"
  }
}
```

sidebar_position: 11 description: The IPC API allows users to create UNIX domain sockets for blockchains to publish to. Find out more information here.

IPC API

The IPC API allows users to create Unix domain sockets for blockchains to publish to. When the blockchain accepts a vertex/block it will publish it to a socket and the decisions contained inside will be published to another.

A node will only expose this API if it is started with [config flag](#) `api-ipcs-enabled=true`.

:::info

This API set is for a specific node, it is unavailable on the [public server](#).

:::

IPC Message Format

Socket messages consist of a 64bit integer in BigEndian format followed by that many bytes.

Example:

```
Sending:
[0x41, 0x76, 0x61, 0x78]
Writes to the socket:
[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x41, 0x76, 0x61, 0x78]
```

IPC Socket URL Format

The names of the sockets are of the form `<network_id>-<chain_id>-<event_type>` where `<event_type>` is either `consensus` or `decisions`. The consensus socket receives vertices and blocks while the decisions socket receives individual transactions.

Format

This API uses the `json 2.0` RPC format.

Endpoint

`/ext/ipcs`

Methods

`ipcs.publishBlockchain`

:::caution

Deprecated as of [v1.9.12](#).

:::

Register a blockchain so it publishes accepted vertices to a Unix domain socket.

Signature:

```
ipcs.publishBlockchain({blockchainID: string}) -> {consensusURL: string, decisionsURL: string}
```

- `blockchainID` is the blockchain that will publish accepted vertices.
 - `consensusURL` is the path of the Unix domain socket the vertices are published to.
 - `decisionsURL` is the path of the Unix domain socket the transactions are published to.

Example Call:

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "decisionsURL": "/tmp/1-11111111111111111111111111111111LpoYY-consensus",  
        "consensusURL": "/tmp/1-11111111111111111111111111111111LpoYY-decisions"  
    },  
    "id": 1  
}
```

`ipcs.unpublishBlockchain`

:::caution

Deprecated as of [v1.9.12](#)

•

Deregister a blockchain so that it no longer publishes to a Unix domain socket.

Signature:

```
ipcs.unpublishBlockchain({blockchainID: string}) -> {}
```

- `blockchainID` is the blockchain that will no longer publish to a Unix domain socket.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "ipcs.unpublishBlockchain",
    "params": {
        "blockchainID": "11111111111111111111111111111111LpoYY"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/ipcs
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {},  
    "id": 1  
}
```

sidebar_position: 2

Issuing API Calls

This guide explains how to make calls to APIs exposed by Avalanche nodes.

Endpoints

An API call is made to an endpoint, which is a URL, made up of the base URI which is the address and the port of the node, and the path the particular endpoint the API call is on.

Base URL

The base of the URL is always:

```
[node-ip] : [http-port]
```

where

- `node-ip` is the IP address of the node the call is to.
- `http-port` is the port the node listens on for HTTP calls. This is specified by [command-line argument](#) `http-port` (default value 9650).

For example, if you're making RPC calls on the local node, the base URL might look like this: `127.0.0.1:9650`.

If you're making RPC calls to remote nodes, then instead of `127.0.0.1` you should use the public IP of the server where the node is. Note that by default the node will only accept API calls on the local interface, so you will need to set up the [`http-host`](#) config flag on the node. Also, you will need to make sure the firewall and/or security policy allows access to the `http-port` from the internet.

:::caution

When setting up RPC access to a node, make sure you don't leave the `http-port` accessible to everyone! There are malicious actors that scan for nodes that have unrestricted access to their RPC port and then use those nodes for spamming them with resource-intensive queries which can knock the node offline. Only allow access to your node's RPC port from known IP addresses!

:::

Endpoint Path

Each API's documentation specifies what endpoint path a user should make calls to in order to access the API's methods.

In general, they are formatted like:

```
/ext/ [api-name]
```

So for the Admin API, the endpoint path is `/ext/admin`, for the Info API it is `/ext/info` and so on. Note that some APIs have additional path components, most notably the chain RPC endpoints which includes the Subnet chain RPCs. We'll go over those in detail in the next section.

So, in combining the base URL and the endpoint path we get the complete URL for making RPC calls. For example, to make a local RPC call on the Info API, the full URL would be:

```
http://127.0.0.1:9650/ext/info
```

Primary Network and Subnet RPC calls

Besides the APIs that are local to the node, like Admin or Metrics APIs, nodes also expose endpoints for talking to particular chains that are either part of the Primary Network (the X, P and C chains), or part of any Subnets the node might be syncing or validating.

In general, chain endpoints are formatted as:

```
ext/bc/ [blockchainID]
```

Primary Network Endpoints

The Primary Network consists of three chains: X, P and C chain. As those chains are present on every node, there are also convenient aliases defined that can be used instead of the full blockchainIDs. So, the endpoints look like:

```
ext/bc/X  
ext/bc/P  
ext/bc/C
```

C-Chain and Subnet-EVM Endpoints

C-Chain and many Subnets run a version of the EthereumVM (EVM). EVM exposes its own endpoints, which are also accessible on the node: JSON-RPC, and Websocket.

JSON-RPC EVM Endpoints

To interact with C-Chain EVM via the JSON-RPC use the endpoint:

```
/ext/bc/C/rpc
```

To interact with Subnet instances of the EVM via the JSON-RPC endpoint:

```
/ext/bc/ [blockchainID]/rpc
```

where `blockchainID` is the ID of the blockchain running the EVM. So for example, the RPC URL for the DFK Network (a Subnet that runs the DeFi Kingdoms:Crystalvale game) running on a local node would be:

```
http://127.0.0.1/ext/bc/q2aTwKuyzgs8pynF7UXBZCU7DejbZbZ6EUyHr3JQzYgwNPUPi/rpc
```

Or for the WAGMI Subnet on the Fuji testnet:

```
http://127.0.0.1/ext/bc/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeqh6tt/rpc
```

WebSocket EVM Endpoints

To interact with C-Chain via the websocket endpoint, use:

```
/ext/bc/C/ws
```

To interact with other instances of the EVM via the websocket endpoint:

```
/ext/bc/blockchainID/ws
```

where `blockchainID` is the ID of the blockchain running the EVM. For example, to interact with the C-Chain's Ethereum APIs via websocket on localhost you can use:

```
ws://127.0.0.1:9650/ext/bc/C/ws
```

```
::info
```

When using the [Public API](#) or another host that supports HTTPS, use `https://` or `wss://` instead of `http://` or `ws://`.

Also, note that the [public API](#) only supports C-Chain websocket API calls for API methods that don't exist on the C-Chain's HTTP API.

```
:::
```

Making a JSON RPC Request

Most of the built-in APIs use the [JSON RPC 2.0](#) format to describe their requests and responses. Such APIs include the Platform API and the X-Chain API.

Suppose we want to call the `getTxStatus` method of the [X-Chain API](#). The X-Chain API documentation tells us that the endpoint for this API is `/ext/bc/X`.

That means that the endpoint we send our API call to is:

```
[node-ip]:[http-port]/ext/bc/X
```

The X-Chain API documentation tells us that the signature of `getTxStatus` is:

```
avm.getTxStatus (txID:bytes) -> (status:string)
```

where:

- Argument `txID` is the ID of the transaction we're getting the status of.
- Returned value `status` is the status of the transaction in question.

To call this method, then:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "avm.getTxStatus",
  "params": {
    "txID": "2QuuvFWUbjuySRxeX5xMbNCuAaKWfbk5FeEa2JmoF85RKLk2dD"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

- `jsonrpc` specifies the version of the JSON RPC protocol. (In practice is always 2.0)
- `method` specifies the service (`avm`) and method (`getTxStatus`) that we want to invoke.
- `params` specifies the arguments to the method.
- `id` is the ID of this request. Request IDs should be unique.

That's it!

JSON RPC Success Response

If the call is successful, the response will look like this:

```
{
  "jsonrpc": "2.0",
  "result": {
    "Status": "Accepted"
  },
}
```

```
        "id": 1
    }

    • id is the ID of the request that this response corresponds to.
    • result is the returned values of getTxStatus.
```

JSON RPC Error Response

If the API method invoked returns an error then the response will have a field `error` in place of `result`. Additionally, there is an extra field, `data`, which holds additional information about the error that occurred.

Such a response would look like:

```
{
    "jsonrpc": "2.0",
    "error": {
        "code": -32600,
        "message": "[Some error message here]",
        "data": [Object with additional information about the error]
    },
    "id": 1
}
```

Other API Formats

Some APIs may use a standard other than JSON RPC 2.0 to format their requests and responses. Such extension should specify how to make calls and parse responses to them in their documentation.

Sending and Receiving Bytes

Unless otherwise noted, when bytes are sent in an API call/response, they are in hex representation.

However, Transaction IDs (TXIDs), ChainIDs, and subnetIDs are in [CB58](#) representation, a base-58 encoding with a checksum.

sidebar_position: 12

Keystore API

:::warning Because the node operator has access to your plain-text password, you should only create a keystore user on a node that you operate. If that node is breached, you could lose all your tokens. Keystore APIs are not recommended for use on Mainnet. :::

Every node has a built-in keystore. Clients create users on the keystore, which act as identities to be used when interacting with blockchains. A keystore exists at the node level, so if you create a user on a node it exists *only* on that node. However, users may be imported and exported using this API.

For validation and cross-chain transfer on the Mainnet, you should issue transactions through [AvalancheJS](#). That way control keys for your funds won't be stored on the node, which significantly lowers the risk should a computer running a node be compromised. See following docs for details:

- [Transfer AVAX Tokens Between Chains](#)
- [Add a Node to the Validator Set](#)

:::info

This API set is for a specific node, it is unavailable on the [public server](#).

:::

Format

This API uses the `json 2.0` API format. For more information on making JSON RPC calls, see [here](#).

Endpoint

```
/ext/keystore
```

Methods

keystore.createUser

:::caution

Deprecated as of [v1.9.12](#).

:::

Create a new user with the specified username and password.

Signature:

```
keystore.createUser(
  {
    username:string,
    password:string
  }
) -> {}
```

- `username` and `password` can be at most 1024 characters.
- Your request will be rejected if `password` is too weak. `password` should be at least 8 characters and contain upper and lower case letters as well as numbers and symbols.

Example Call:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" :1,
  "method" :"keystore.createUser",
  "params" :{
    "username":"myUsername",
    "password":"myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/keystore
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

keystore.deleteUser

:::caution

Deprecated as of [v1.9.12](#).

:::

Delete a user.

Signature:

```
keystore.deleteUser({username: string, password:string}) -> {}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" :1,
  "method" :"keystore.deleteUser",
  "params" : {
    "username":"myUsername",
    "password":"myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/keystore
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

keystore.exportUser

:::caution

Deprecated as of [v1.9.12](#).

:::

Export a user. The user can be imported to another node with [keystore.importUser](#). The user's password remains encrypted.

Signature:

```

keystore.exportUser(
  {
    username:string,
    password:string,
    encoding:string //optional
  }
) -> {
  user:string,
  encoding:string
}

```

`encoding` specifies the format of the string encoding user data. Can only be `hex` when a value is provided.

Example Call:

```

curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" :1,
  "method" :"keystore.exportUser",
  "params" :{
    "username":"myUsername",
    "password":"myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/keystore

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "user": "7655a29df6fc2747b0874e1148b423b954a25fcdb1f170d0ec8eb196430f7001942ce55b02a83b1faf50a674b1e55bfc00000000",
    "encoding": "hex"
  }
}
```

keystore.importUser

::caution

Deprecated as of [v1.9.12](#).

:::

Import a user. `password` must match the user's password. `username` doesn't have to match the username `user` had when it was exported.

Signature:

```

keystore.importUser(
  {
    username:string,
    password:string,
    user:string,
    encoding:string //optional
  }
) -> {}

```

`encoding` specifies the format of the string encoding user data. Can only be `hex` when a value is provided.

Example Call:

```

curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" :1,
  "method" :"keystore.importUser",
  "params" :{
    "username":"myUsername",
    "password":"myPassword",
    "user":":0x7655a29df6fc2747b0874e1148b423b954a25fcdb1f170d0ec8eb196430f7001942ce55b02a83b1faf50a674b1e55bfc000000008cf2d869"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/keystore

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {}
}
```

keystore.listUsers

:::caution

Deprecated as of [v1.9.12](#).

:::

List the users in this keystore.

Signature:

```
keystore.ListUsers() -> {users:[]string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "keystore.listUsers"
}' -H 'content-type:application/json' 127.0.0.1:9650/ext/keystore
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "users": ["myUsername"]
  }
}
```

sidebar_position: 13

Metrics API

The API allows clients to get statistics about a node's health and performance.

:::info

This API set is for a specific node, it is unavailable on the [public server](#).

:::

Endpoint

```
/ext/metrics
```

Usage

To get the node metrics:

```
curl -X POST 127.0.0.1:9650/ext/metrics
```

Format

This API produces Prometheus compatible metrics. See [here](#) for information on Prometheus' formatting.

[Here](#) is a tutorial that shows how to set up Prometheus and Grafana to monitor AvalancheGo node using the Metrics API.

sidebar_position: 3

Platform Chain (P-Chain) API

This API allows clients to interact with the [P-Chain](#), which maintains Avalanche's [validator](#) set and handles blockchain creation.

Endpoint

```
/ext/bc/p
```

Format

This API uses the `json 2.0` RPC format.

Methods

```
platform.addDelegator
```

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Add a delegator to the Primary Network.

A delegator stakes AVAX and specifies a validator (the delegatee) to validate on their behalf. The delegatee has an increased probability of being sampled by other validators (weight) in proportion to the stake delegated to them.

The delegatee charges a fee to the delegator; the former receives a percentage of the delegator's validation reward (if any.) A transaction that delegates stake has no fee.

The delegation period must be a subset of the period that the delegatee validates the Primary Network.

:::info

Once you issue the transaction to add a node as a delegator, there is no way to change the parameters. **You can't remove a stake early or change the stake amount, node ID, or reward address.** Please make sure you're using the correct values. If you're not sure, please reach out to us on [Discord](#).

:::

Signature:

```
platform.addDelegator(  
  {  
    nodeID: string,  
    startTime: int,  
    endTime: int,  
    stakeAmount: int,  
    rewardAddress: string,  
    from: []string, // optional  
    changeAddr: string, // optional  
    username: string,  
    password: string  
  }  
) ->  
{  
  txID: string,  
  changeAddr: string  
}
```

- `nodeID` is the ID of the node to delegate to.
- `startTime` is the Unix time when the delegator starts delegating.
- `endTime` is the Unix time when the delegator stops delegating (and staked AVAX is returned).
- `stakeAmount` is the amount of nAVAX the delegator is staking.
- `rewardAddress` is the address the validator reward goes to, if there is one.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user that pays the transaction fee.
- `password` is `username`'s password.
- `txID` is the transaction ID

Example Call:

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "method": "platform.addDelegator",
```

```

"params": {
    "nodeID": "NodeID-MFrZFVCXPv5iCn6M9K6XduxGTYp891xXZ",
    "rewardAddress": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "startTime": 1594102400,
    "endTime": 1604102400,
    "stakeAmount": 100000,
    "from": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u",
    "username": "myUsername",
    "password": "myPassword"
},
"id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "6pB3MtHUNogeHapZqMUBmx6N38ii3LzytVDrXuMovwKQFTZLs",
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u"
  },
  "id": 1
}
```

platform.addSubnetValidator

::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Add a validator to a Subnet other than the Primary Network. The Validator must validate the Primary Network for the entire duration they validate this Subnet.

Signature:

```
platform.addSubnetValidator(
  {
    nodeID: string,
    subnetID: string,
    startTime: int,
    endTime: int,
    weight: int,
    from: []string, // optional
    changeAddr: string, // optional
    username: string,
    password: string
  }
) ->
{
  txID: string,
  changeAddr: string,
}
```

- `nodeID` is the node ID of the validator being added to the Subnet. This validator must validate the Primary Network for the entire duration that it validates this Subnet.
- `subnetID` is the ID of the Subnet we're adding a validator to.
- `startTime` is the Unix time when the validator starts validating the Subnet. It must be at or after the time that the validator starts validating the Primary Network.
- `endTime` is the Unix time when the validator stops validating the Subnet. It must be at or before the time that the validator stops validating the Primary Network.
- `weight` is the validator's weight used for sampling. If the validator's weight is 1 and the cumulative weight of all validators in the Subnet is 100, then this validator will be included in about 1 in every 100 samples during consensus. The cumulative weight of all validators in the Subnet must be at least `snow-sample-size`. For example, if there is only one validator in the Subnet, its weight must be at least `snow-sample-size` (default 20). Recall that a validator's weight can't be changed while it is validating, so take care to use an appropriate value.
- `from` are the fund addresses that the user wants to use to pay for this operation. If omitted, use any of user's addresses as needed.
- `changeAddr` is the address any change/left-over of the fund (specified by the `from` addresses) will be sent to. If omitted, change/left-over is sent to one of the addresses controlled by the user.
- `username` is the user that pays the transaction fee.
- `password` is `username`'s password.

- `txID` is the transaction ID.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.addSubnetValidator",
  "params": {
    "nodeID": "NodeID-7xhw2mdxuds44j42tcb6u5579esbst3lg",
    "subnetID": "zbffowlffkpvrfywpj1cvqrfnyesepdfc61hmu2n9jnghduel",
    "startTime": 1583524047,
    "endTime": 1604102399,
    "weight": 1,
    "from": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnxux8r73r8u",
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2exafyvRNSE5ehwjhafBvt6CTntot7DFjsZNCz54GSxBbVLcM",
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnxux8r73r8u"
  }
}
```

platform.addValidator

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Add a validator to the Primary Network. You must stake AVAX to do this. If the node is sufficiently correct and responsive while validating, you receive a reward when end of staking period is reached. The validator's probability of being sampled by other validators during consensus is in proportion to the amount of AVAX staked.

The validator charges a fee to delegators; the former receives a percentage of the delegator's validation reward (if any.) The minimum delegation fee is 2%. A transaction that adds a validator has no fee.

The validation period must be between 2 weeks and 1 year for the Mainnet, and 24 hours and 1 year for Fuji Testnet.

There is a maximum total weight imposed on validators. This means that no validator will ever have more AVAX staked and delegated to it than this value. This value will initially be set to `min(5 * amount staked, 3M AVAX)`. The total value on a validator is 3 million AVAX.

:::note

Note that once you issue the transaction to add a node as a validator, there is no way to change the parameters. **You can't remove stake early or change the stake amount, node ID, or reward address.** Please make sure you're using the correct values. If you're not sure, please reach out to us on [Discord](#).

:::

Signature:

```
platform.addValidator(
  {
    nodeID: string,
    startTime: int,
    endTime: int,
    stakeAmount: int,
    rewardAddress: string,
    delegationFeeRate: float,
    from: []string, // optional
    changeAddr: string, // optional
    username: string,
    password: string
  }
) ->
```

```
{
  txID: string,
  changeAddr: string
}
```

- `nodeID` is the node ID of the validator being added.
- `startTime` is the Unix time when the validator starts validating the Primary Network.
- `endTime` is the Unix time when the validator stops validating the Primary Network (and staked AVAX is returned).
- `stakeAmount` is the amount of nAVAX the validator is staking.
- `rewardAddress` is the address the validator reward will go to, if there is one.
- `delegationFeeRate` is the percent fee this validator charges when others delegate stake to them. Up to 4 decimal places allowed; additional decimal places are ignored. Must be between 0 and 100, inclusive. For example, if `delegationFeeRate` is 1.2345 and someone delegates to this validator, then when the delegation period is over, 1.2345% of the reward goes to the validator and the rest goes to the delegator.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user that pays the transaction fee.
- `password` is `username`'s password.
- `txID` is the transaction ID

Example Call:

In this example, we use shell command `date` to compute Unix times 10 minutes and 2 days in the future. (Note: If you're on a Mac, replace `$(date)` with `$(gdate)`. If you don't have `gdate` installed, do `brew install coreutils`.)

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.addValidator",
  "params": {
    "nodeID": "NodeID=ARCLrphAHZ28xZEBfUL7SVAmzkTZNelLK",
    "rewardAddress": "P-avax18jma8pw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "from": ["P-avax18jma8pw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u",
    "startTime":'$(date --date="10 minutes" +%)',
    "endTime":'$(date --date="2 days" +%)',
    "stakeAmount":1000000,
    "delegationFeeRate":10,
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "6pb3mthunogehapzqubmx6n38ii3lzytvdrxumovwkqftzls",
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u"
  },
  "id": 1
}
```

platform.createAddress

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Create a new address controlled by the given user.

Signature:

```
platform.createAddress ({
  username: string,
  password: string
}) -> {address: string}
```

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.createAddress",
    "params": {
        "username": "myUsername",
        "password": "myPassword"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/F
```

Example Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    },
    "id": 1
}
```

```
platform.createBlockchain
```

:::caution

Deprecated as of [v1.9.12](#).

• • •
• • •

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#)

20

Create a new blockchain. Currently only supports the creation of new instances of the AVM and the Timestamp VM.

Signature:

```
platform.createBlockchain(
{
    subnetID: string,
    vmID: string,
    name: string,
    genesisData: string,
    encoding: string, // optional
    from: [string, // optional
    changeAddr: string, // optional
    username: string,
    password: string
}
) ->
{
    txID: string,
    changeAddr: string
}
}
```

- `subnetID` is the ID of the Subnet that validates the new blockchain. The Subnet must exist and can't be the Primary Network.
 - `vmID` is the ID of the Virtual Machine the blockchain runs. Can also be an alias of the Virtual Machine.
 - `name` is a human-readable name for the new blockchain. Not necessarily unique.
 - `genesisData` is the byte representation of the genesis state of the new blockchain encoded in the format specified by the `encoding` parameter.
 - `encoding` specifies the format to use for `genesisData`. Can only be `hex` when a value is provided. Virtual Machines should have a static API method named `buildGenesis` that can be used to generate `genesisData`
 - `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
 - `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
 - `username` is the user that pays the transaction fee. This user must have a sufficient number of the Subnet's control keys.
 - `password` is `username`'s password.
 - `txID` is the transaction ID.

Example Call:

In this example we're creating a new instance of the Timestamp Virtual Machine. `genesisData` came from calling `timestamp.buildGenesis()`.

```
curl -X POST --data '{  
    "jsonrpc": "2.0",  
    "method": "platform.createBlockchain",  
    "params" : {  
        "vmID": "timestamp".
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "txID": "2TBnyFmST7TirNm6Y6z4863zusRVpWi5CjlsKS9bXTUmu8GfeU",  
        "changeAddr": "P-avaxl03y30cxeu1kjfe3kwfnpt432ylmnux8r73r8u"  
    },  
    "id": 1  
}
```

```
platform.createSubnet
```

:::caution

Deprecated as of [v1.9.12](#).

•

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#)

* * *

Create a new Subnet.

The Subnet's ID is the same as this transaction's ID

Signature:

```
platform.createSubnet(
{
    controlKeys: []string,
    threshold: int,
    from: []string, // optional
    changeAddr: string, // optional
    username: string,
    password: string
}
) ->
{
    txID: string,
    changeAddr: string
}
```

- In order to add a validator to this Subnet, `threshold` signatures are required from the addresses in `controlKeys`
 - `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
 - `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
 - `username` is the user that pays the transaction fee.
 - `password` is `username`'s password.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.createSubnet",
    "params": {
        "controlKeys": [
            "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
            "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"
        ],
        "threshold": 2,
        "from": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
        "changeAddr": "P-avax103y30cxelujkfe3kwfnpt432ylmnux8r73r8u",
        "username": "myUsername",
        "password": "myPassword"
    }
}'
```

```
},
  "id": 1
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "hJfc5xGhtjhCGBh1JWn3vZ51KJP696TzrsbadPHNbQG2Ve5yd"
  },
  "id": 1
}
```

platform.exportAVAX

::caution

Deprecated as of [v1.9.12](#).

:::

Send AVAX from an address on the P-Chain to an address on the X-Chain or C-Chain. After issuing this transaction, you must call the X-Chain's [avm.import](#) or C-Chain's [avax.import](#) with assetID `AVAX` to complete the transfer.

Signature:

```
platform.exportAVAX(
  {
    amount: int,
    from: []string, // optional
    to: string,
    changeAddr: string, // optional
    username: string,
    password: string
  }
) ->
{
  txID: string,
  changeAddr: string
}
```

- `amount` is the amount of nAVAX to send.
- `to` is the address on the X-Chain or C-Chain to send the AVAX to.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user sending the AVAX and paying the transaction fee.
- `password` is `username`'s password.
- `txID` is the ID of this transaction.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.exportAVAX",
  "params": {
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "amount": 1,
    "from": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u",
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "2Kz69TNBSeABuaVjKa6ZJCTLobbe5xo9c5eU8QwdUSvPo2dBk3",
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u"
  },
  "id": 1
}
```

`platform.exportKey`

::caution

Deprecated as of [v1.9.12](#).

::

::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

::

Get the private key that controls a given address. The returned private key can be added to a user with `platform.importKey`.

Signature:

```
platform.exportKey({
  username: string,
  password: string,
  address: string
}) -> {privateKey: string}
```

- `username` is the user that controls `address`.
- `password` is `username`'s password.
- `privateKey` is the string representation of the private key that controls `address`.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.exportKey",
  "params": {
    "username": "myUsername",
    "password": "myPassword",
    "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "privateKey": "PrivateKey-Lf49kAJw3CbaL783vmbeAJvhscJqC7vi5yBYLxw2XfbzNS5RS"
  }
}
```

`platform.getBalance`

::caution

Deprecated as of [v1.9.12](#).

::

Get the balance of AVAX controlled by a given address.

Signature:

```
platform.getBalance({
  addresses: []string
}) -> {
  balances: string -> int,
  unlockeds: string -> int,
  lockedStakeables: string -> int,
  lockedNotStakeables: string -> int,
  utxoIDs: []{
    txID: string,
    outputIndex: int
  }
}
```

- `addresses` are the addresses to get the balance of.
- `balances` is a map from assetID to the total balance.
- `unlockeds` is a map from assetID to the unlocked balance.

- `lockedStakeables` is a map from assetID to the locked stakeable balance.
 - `lockedNotStakeables` is a map from assetID to the locked and not stakeable balance.
 - `utxoIDs` are the IDs of the UTXOs that reference `address`.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id" : 1,
    "method" :"platform.getBalance",
    "params" :{
        "addresses": ["P-custom18jma8ppw3nhx5r4ap8clazz0dps7rv5u9xde7p"],
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "balance": "30000000000000000000",
        "unlocked": "20000000000000000000",
        "lockedStakeable": "10000000000000000000",
        "lockedNotStakeable": "0",
        "balances": {
            "BUuypiq2wyuLMvyhzFXcPyxPMcGSp7eeDohhQRqTChoBjKziC": "30000000000000000000"
        },
        "unlockeds": {
            "BUuypiq2wyuLMvyhzFXcPyxPMcGSp7eeDohhQRqTChoBjKziC": "20000000000000000000"
        },
        "lockedStakeables": {
            "BUuypiq2wyuLMvyhzFXcPyxPMcGSp7eeDohhQRqTChoBjKziC": "10000000000000000000"
        },
        "lockedNotStakeables": {},
        "utxoIDs": [
            {
                "txID": "11111111111111111111111111111111LpoYY",
                "outputIndex": 1
            },
            {
                "txID": "11111111111111111111111111111111LpoYY",
                "outputIndex": 0
            }
        ]
    },
    "id": 1
}
```

platform.getBlock

Get a block by its ID.

Signature:

```
platform.getBlock({
  blockID: string
  encoding: string // optional
}) -> {
  block: string,
  encoding: string
}
```

Request:

- `blockID` is the block ID. It should be in cb58 format.
 - `encoding` is the encoding format to use. Can be either `hex` or `json`. Defaults to `hex`

Response:

- `block` is the transaction encoded to `encoding`.
 - `encoding` is the `encoding`.

Hex Example

Example Call:

```
curl -X POST --data '{  
    "jsonrpc": "2.0",
```

```
"method": "platform.getBlock",
"params": {
    "blockID": "d7WYmb8VeZNHsny3EJCwMm6QA37s1EHwMxw1Y71V3FqPz5EFG",
    "encoding": "hex"
},
"id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "block":  
"0x000000000000309473dc99a0851a29174d84e522da8ccb1a56ac23f7b0ba79f80acce34cf5769000000000000f42410000001000000120000001000000000000  
        "encoding": "hex"  
    },  
    "id": 1  
}
```

JSON Example

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getBlock",
    "params": {
        "blockID": "d7WYmb8VeZNHsny3EJcWmM6QA37s1EHwMxw1Y71V3FqPz5EFG",
        "encoding": "json"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```

        ]
    },
    "credentials": [
        {
            "signatures": [
                "0xc79711c4b48dcde205b63603efef7c61773a0eb47efb503fcebe40d21962b7c25ebd734057400a12cce9cf99aceec8462923d5d91ffffe1cb908372281ed7385"
            ]
        }
    ]
},
"encoding": "json"
},
"id": 1
}

```

`platform.getBlockchains`

::caution

Deprecated as of [v1.9.12](#).

:::

Get all the blockchains that exist (excluding the P-Chain).

Signature:

```

platform.getBlockchains() ->
{
  blockchains: []{
    id: string,
    name:string,
    subnetID: string,
    vmID: string
  }
}

```

- `blockchains` is all of the blockchains that exists on the Avalanche network.
- `name` is the human-readable name of this blockchain.
- `id` is the blockchain's ID.
- `subnetID` is the ID of the Subnet that validates this blockchain.
- `vmID` is the ID of the Virtual Machine the blockchain runs.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getBlockchains",
  "params": {},
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "blockchains": [
      {
        "id": "2oYMBNV4eNHqk2fjjV5nVQLDbtmNzq5s3qs3Lo6ftnC6FBYm",
        "name": "X-Chain",
        "subnetID": "11111111111111111111111111111111LpoYY",
        "vmID": "jvYyfQTxGMJLuGWA55kdP2p2zSUysQ5Raupu4TW34ZAUBAbtq"
      },
      {
        "id": "2q9e4r6Mu3U68nU1fyjgbR6JvwrRx36CohpAX5UQxse55x1Q5",
        "name": "C-Chain",
        "subnetID": "11111111111111111111111111111111LpoYY",
        "vmID": "mgj786NP7uDwBCcq6YwThhaN8FLyybkCa4zBWTQbNgmK6k9A6"
      },
      {
        "id": "CqhF97NNugqYLiGaQJ2xckfmkEr8uNeGG5TQbyGcgNZ5ahQwa",
        "name": "Simple DAG Payments",
        "subnetID": "11111111111111111111111111111111LpoYY",
        "vmID": "sgjdyTKUSRQs1YmKDTUbduhdstSdtRTGRbUn8sqK8B6pkZkz1"
      }
    ]
  }
}
```

```
platform.getBlockchainStatus
```

Get the status of a blockchain.

Signature:

```
platform.getBlockchainStatus()
{
    blockchainID: string
}
) -> {status: string}
```

status is one of:

- **Validating** : The blockchain is being validated by this node.
 - **Created** : The blockchain exists but isn't being validated by this node.
 - **Preferred** : The blockchain was proposed to be created and is likely to be created but the transaction isn't yet accepted.
 - **Syncing** : This node is participating in this blockchain as a non-validating node.
 - **Unknown** : The blockchain either wasn't proposed or the proposal to create it isn't preferred. The proposal may be resubmitted.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getBlockchainStatus",
    "params": {
        "blockchainID": "2NbS4dwGaf2p1MaXb65PrkZdXRwmSX4ZzGnUu7jm3aykgThuZE",
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "status": "Created"  
  },  
  "id": 1  
}
```

```
platform.getCurrentSupply
```

Returns an upper bound on amount of tokens that exist that can stake the requested Subnet. This is an upper bound because it does not account for burnt tokens, including transaction fees.

Signature:

```
platform.getCurrentSupply({  
    subnetID: string // optional  
}) -> {supply: int}
```

- `supply` is an upper bound on the number of tokens that exist.

Example Call:

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "supply": "365865167637779183"  
  },  
  "id": 1  
}
```

The response in this example indicates that AVAX's supply is at most 365.865 million.

```
platform.getCurrentValidators
```

List the current validators of the given Subnet

Signature:

```
platform.getCurrentValidators({
    subnetID: string, // optional
    nodeIDs: string[], // optional
}) -> {
    validators: [](
        txID: string,
        startTime: string,
        endTime: string,
        stakeAmount: string,
        nodeID: string,
        weight: string,
        validationRewardOwner: {
            locktime: string,
            threshold: string,
            addresses: string[]
        },
        delegationRewardOwner: {
            locktime: string,
            threshold: string,
            addresses: string[]
        },
        potentialReward: string,
        delegationFee: string,
        uptime: string,
        connected: bool,
        signer: {
            publicKey: string,
            proofOfPosession: string
        },
        delegatorCount: string,
        delegatorWeight: string,
        delegators: [](
            txID: string,
            startTime: string,
            endTime: string,
            stakeAmount: string,
            nodeID: string,
            rewardOwner: {
                locktime: string,
                threshold: string,
                addresses: string[]
            },
        ),
    ),
}
```

```

        potentialReward: string,
    }
}
}

• subnetID is the Subnet whose current validators are returned. If omitted, returns the current validators of the Primary Network.
• nodeIDs is a list of the NodeIDs of current validators to request. If omitted, all current validators are returned. If a specified NodeID is not in the set of current validators, it will not be included in the response.
• validators:


- txID is the validator transaction.
- startTime is the Unix time when the validator starts validating the Subnet.
- endTime is the Unix time when the validator stops validating the Subnet.
- stakeAmount is the amount of tokens this validator staked. Omitted if subnetID is not a PoS Subnet.
- nodeID is the validator's node ID.
- weight is the validator's weight when sampling validators. Omitted if subnetID is a PoS Subnet.
- validationRewardOwner is an OutputOwners output which includes locktime, threshold and array of addresses. Specifies the owner of the potential reward earned from staking. Omitted if subnetID is not a PoS Subnet.
- delegationRewardOwner is an OutputOwners output which includes locktime, threshold and array of addresses. Specifies the owner of the potential reward earned from delegations. Omitted if subnetID is not a PoS Subnet.
- potentialReward is the potential reward earned from staking. Omitted if subnetID is not a PoS Subnet.
- delegationFeeRate is the percent fee this validator charges when others delegate stake to them. Omitted if subnetID is not a PoS Subnet.
- uptime is the % of time the queried node has reported the peer as online and validating the Subnet. Omitted if subnetID is not a PoS Subnet.
- connected is if the node is connected and tracks the Subnet.
- signer is the node's BLS public key and proof of possession. Omitted if the validator doesn't have a BLS public key.
- delegatorCount is the number of delegators on this validator. Omitted if subnetID is not a PoS Subnet.
- delegatorWeight is total weight of delegators on this validator. Omitted if subnetID is not a PoS Subnet.
- delegators is the list of delegators to this validator. Omitted if subnetID is not a PoS Subnet. Omitted unless nodeIDs specifies a single NodeID.
  - txID is the delegator transaction.
  - startTime is the Unix time when the delegator started.
  - endTime is the Unix time when the delegator stops.
  - stakeAmount is the amount of nAVAX this delegator staked.
  - nodeID is the validating node's node ID.
  - rewardOwner is an OutputOwners output which includes locktime, threshold and array of addresses.
  - potentialReward is the potential reward earned from staking

```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getCurrentValidators",
  "params": {
    "nodeIDs": ["NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD"]
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "validators": [
      {
        "txID": "2NNkpYTGftTfLSGXJcHtVv6drwVU2cczhmjK2uhvwDyxwsjzZMm",
        "startTime": "1600368632",
        "endTime": "1602960455",
        "stakeAmount": "200000000000",
        "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD",
        "validationRewardOwner": {
          "locktime": "0",
          "threshold": "1",
          "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"]
        },
        "delegationRewardOwner": {
          "locktime": "0",
          "threshold": "1",
          "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"]
        },
        "potentialReward": "117431493426",
        "delegationFee": "10.0000",
        "uptime": "0.0000",
        "connected": false
      }
    ]
  }
}
```

```

        "delegatorCount": "1",
        "delegatorWeight": "25000000000",
        "delegators": [
            {
                "txID": "Bbai8nzGVcyn2VmeYcbS74zfjJLjDacGNVuzuvAQkHnluWfov",
                "startTime": "1600368523",
                "endTime": "1602960342",
                "stakeAmount": "25000000000",
                "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD",
                "rewardOwner": {
                    "locktime": "0",
                    "threshold": "1",
                    "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"]
                },
                "potentialReward": "11743144774"
            }
        ]
    },
    "id": 1
}

```

platform.getHeight

Returns the height of the last accepted block.

Signature:

```

platform.getHeight() ->
{
    height: int,
}

```

Example Call:

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getHeight",
    "params": {},
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

Example Response:

```

{
    "jsonrpc": "2.0",
    "result": {
        "height": "56"
    },
    "id": 1
}

```

platform.getMaxStakeAmount

:::caution

Deprecated as of [v1.9.12](#).

:::

Returns the maximum amount of nAVAX staking to the named node during a particular time period.

Signature:

```

platform.getMaxStakeAmount(
    {
        subnetID: string,
        nodeID: string,
        startTime: int,
        endTime: int
    }
) ->
{
    amount: uint64
}

```

- `subnetID` is a Buffer or cb58 string representing Subnet
- `nodeID` is a string representing ID of the node whose stake amount is required during the given duration
- `startTime` is a big number denoting start time of the duration during which stake amount of the node is required.
- `endTime` is a big number denoting end time of the duration during which stake amount of the node is required.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getMaxStakeAmount",
  "params": {
    "subnetID": "11111111111111111111111111111111LpoYY",
    "nodeID": "NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg",
    "startTime": 1644240334,
    "endTime": 1644240634
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "amount": "2000000000000000"
  },
  "id": 1
}
```

platform.getMinStake

Get the minimum amount of tokens required to validate the requested Subnet and the minimum amount of tokens that can be delegated.

Signature:

```
platform.getMinStake({
  subnetID: string // optional
}) ->
{
  minValidatorStake : uint64,
  minDelegatorStake : uint64
}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.getMinStake",
  "params": {
    "subnetID": "11111111111111111111111111111111LpoYY"
  },
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "minValidatorStake": "200000000000",
    "minDelegatorStake": "25000000000"
  },
  "id": 1
}
```

platform.getPendingValidators

List the validators in the pending validator set of the specified Subnet. Each validator is not currently validating the Subnet but will in the future.

Signature:

```
platform.getPendingValidators({
  subnetID: string, // optional
  nodeIDs: string[], // optional
}) -> {
  validators: []
}
```

```

        txID: string,
        startTime: string,
        endTime: string,
        stakeAmount: string,
        nodeID: string,
        delegationFee: string,
        connected: bool,
        signer: {
            publicKey: string,
            proofOfPossession: string
        },
        weight: string,
    },
    delegators: []{
        txID: string,
        startTime: string,
        endTime: string,
        stakeAmount: string,
        nodeID: string
    }
}
}

```

- `subnetID` is the Subnet whose current validators are returned. If omitted, returns the current validators of the Primary Network.
- `nodeIDs` is a list of the NodeIDs of pending validators to request. If omitted, all pending validators are returned. If a specified NodeID is not in the set of pending validators, it will not be included in the response.
- `validators`:
 - `txID` is the validator transaction.
 - `startTime` is the Unix time when the validator starts validating the Subnet.
 - `endTime` is the Unix time when the validator stops validating the Subnet.
 - `stakeAmount` is the amount of tokens this validator staked. Omitted if `subnetID` is not a PoS Subnet.
 - `nodeID` is the validator's node ID.
 - `connected` if the node is connected and tracks the Subnet.
 - `signer` is the node's BLS public key and proof of possession. Omitted if the validator doesn't have a BLS public key.
 - `weight` is the validator's weight when sampling validators. Omitted if `subnetID` is a PoS Subnet.
- `delegators`:
 - `txID` is the delegator transaction.
 - `startTime` is the Unix time when the delegator starts.
 - `endTime` is the Unix time when the delegator stops.
 - `stakeAmount` is the amount of tokens this delegator staked.
 - `nodeID` is the validating node's node ID.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getPendingValidators",
    "params": {},
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
    "jsonrpc": "2.0",
    "result": {
        "validators": [
            {
                "txID": "2NNkpYTGfTFLSGXJcHtVv6drwVu2cczhmjK2uhvwDyxwsjzzMm",
                "startTime": "1600368632",
                "endTime": "1602960455",
                "stakeAmount": "200000000000",
                "nodeID": "NodeID-5mb6qkSBj81k9g9e4VFjGGSbaaSLFRzD",
                "delegationFee": "10.0000",
                "connected": false
            }
        ],
        "delegators": [
            {
                "txID": "Bbai8nzGVcyn2VmeycbS74zfjJLjDacGNVuzuvAQkhnlwWfoV",
                "startTime": "1600368523",
                "endTime": "1602960342",
                "stakeAmount": "20000000000",
                "nodeID": "NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg"
            }
        ]
}
```

```
},
  "id": 1
}
```

platform.getRewardUTXOs

:::caution

Deprecated as of [v1.9.12](#).

:::

Returns the UTXOs that were rewarded after the provided transaction's staking or delegation period ended.

Signature:

```
platform.getRewardUTXOs({
  txID: string,
  encoding: string // optional
}) -> {
  numFetched: integer,
  utxos: []string,
  encoding: string
}
```

- `txID` is the ID of the staking or delegating transaction
- `numFetched` is the number of returned UTXOs
- `utxos` is an array of encoded reward UTXOs
- `encoding` specifies the format for the returned UTXOs. Can only be `hex` when a value is provided.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getRewardUTXOs",
  "params": {
    "txID": "2nmH8LithVbdjaXsxVQCQfXtzN9hBbmebrsaEYnLM9T32Uy2Y5"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "2",
    "utxos": [
      "0x0000a195046108a85e60f7a864bb567745a37f50c6af282103e47cc62f036cee404700000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786",
      "0x0000ae8b1b94444eed8de9a81b1222f00f1b4133330add23d8ac288bffa98b85271100000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786"
    ],
    "encoding": "hex"
  },
  "id": 1
}
```

platform.getStake

:::caution

Deprecated as of [v1.9.12](#).

:::

Get the amount of nAVAX staked by a set of addresses. The amount returned does not include staking rewards.

Signature:

```
platform.getStake({
  addresses: []string,
  validatorsOnly: true or false
}) ->
{
  stakeds: string -> int,
  stakedOutputs: []string,
```

- `addresses` are the addresses to get information about.
- `validatorsonly` can be either `true` or `false`. If `true`, will skip checking delegators for stake.
- `staked` is a map from assetID to the amount staked by addresses provided.
- `stakedOutputs` are the string representation of staked outputs.
- `encoding` specifies the format for the returned outputs.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getStake",
    "params": {
        "addresses": [
            "P-avax1pmgmgcj1jjzuz2ve339dx82khm7q8getlegte"
        ],
        "validatorsOnly": true
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
platform.getStakingAssetID
```

Retrieve an assetID for a Subnet's staking asset.

Signature:

```
platform.getStakingAssetID({
    subnetID: string // optional
}) -> {
    assetID: string
}
```

- `subnetID` is the Subnet whose assetID is requested.
 - `assetID` is the assetID for a Subnet's staking asset.

Example Call:

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "assetID": "2fombhL7aGPwj3KH4bfmJwW6PVnMobf9Y2fn9GwxIAAJyFDbe"  
  }  
}
```

```
        "id": 1
    }
```

:::note

The AssetID for AVAX differs depending on the network you are on.

Mainnet: FvwEAhmxFfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z

Testnet: U8IRqJoiJm8xZHAacmvYyZVwqQx6uDNTQeP3CQ6fcgQk3JqnK

:::

```
platform.getSubnets
```

:::caution

Deprecated as of [v1.9.12](#).

:::

Get info about the Subnets.

Signature:

```
platform.getSubnets({
  ids: []string
}) ->
{
  subnets: []{
    id: string,
    controlKeys: []string,
    threshold: string
  }
}
```

- `ids` are the IDs of the Subnets to get information about. If omitted, gets information about all Subnets.
- `id` is the Subnet's ID.
- `threshold` signatures from addresses in `controlKeys` are needed to add a validator to the Subnet. If the Subnet is a PoS Subnet, then `threshold` will be 0 and `controlKeys` will be empty.

See [here](#) for information on adding a validator to a Subnet.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getSubnets",
  "params": {"ids": ["hW8Ma7dLMA7o4xmJf3AXBbo17bXzE7xnThUd3ypM4VAWo1sNJ"]},
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "subnets": [
      {
        "id": "hW8Ma7dLMA7o4xmJf3AXBbo17bXzE7xnThUd3ypM4VAWo1sNJ",
        "controlKeys": [
          "KNjXsaAlsZsaKCD1cd85YXauDuxshTes2",
          "Aiz4eEt5xv9t4NCnAWaQJFNz5ABqLtJkR"
        ],
        "threshold": "2"
      }
    ],
    "id": 1
  }
}
```

```
platform.getTimestamp
```

Get the current P-Chain timestamp.

Signature:

```
platform.getTimestamp() -> {time: string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getTimestamp",
  "params": {},
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "timestamp": "2021-09-07T00:00:00-04:00"
  },
  "id": 1
}
```

platform.getTotalStake

Get the total amount of tokens staked on the requested Subnet.

Signature:

```
platform.getTotalStake({
  subnetID: string
}) -> {
  stake: int
  weight: int
}
```

Primary Network Example**Example Call:**

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getTotalStake",
  "params": {
    "subnetID": "11111111111111111111111111111111LpoYY"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "stake": "279825917679866811",
    "weight": "279825917679866811"
  },
  "id": 1
}
```

Subnet Example**Example Call:**

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getTotalStake",
  "params": {
    "subnetID": "2bRCr6B4MiEfSjidDwxDpdCyyiwnfUVqB2HGwhm947w9YYqb7r"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "weight": "100000"
  },
  "id": 1
}
```

`platform.getTx`

Gets a transaction by its ID.

Optional `encoding` parameter to specify the format for the returned transaction. Can be either `hex` or `json`. Defaults to `hex`.

Signature:

```
platform.getTx({
  txID: string,
  encoding: string // optional
}) -> {
  tx: string,
  encoding: string,
}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getTx",
  "params": {
    "txID": "28KVjSw5h3XKGujNpJXWY74EdnGq4TUWvCgEtJFymgQTvudiugb",
    "encoding": "json"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "tx": {
      "unsignedTx": {
        "networkID": 1,
        "blockchainID": "11111111111111111111111111111111LpoYY",
        "outputs": [],
        "inputs": [
          {
            "txID": "NXNJHKeaJyjjWVSq341t6LGQP5UNz796c1crpHPByv1TKp9ZP",
            "outputIndex": 0,
            "assetID": "FvwEAhmxFfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
            "fxID": "spdxUxVJqbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEPM3h6TLuxQQ",
            "input": {
              "amount": 20824279595,
              "signatureIndices": [
                0
              ]
            }
          },
          {
            "txID": "2ahK5SzD8iqi5KBqpKfxrnWtrEoVwQCqJsMoB9kvChCaHgAQC9",
            "outputIndex": 1,
            "assetID": "FvwEAhmxFfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
            "fxID": "spdxUxVJqbX85MGxMHbKwlsHxMnSqJ3QBzDyDYEPM3h6TLuxQQ",
            "input": {
              "amount": 28119890783,
              "signatureIndices": [
                0
              ]
            }
          }
        ],
        "memo": "0x",
        "validator": {
          "nodeID": "NodeID-VT3YhgFaWEzy4Ap937qMeNEDscCammzG",
          "start": 1682945406,
        }
      }
    }
  }
}
```

```

        "end": 1684155006,
        "weight": 48944170378
    },
    "stake": [
        {
            "assetID": "FvwEAhmxFfeiG8SnEvq42hc6whRyY3EFYAvbMqDNDGcxN5Z",
            "fxID": "spdxUxVJQbx85MGxMHbKw1shHxMnSqJ3QBzDyDYEP3h6TLuxqQ",
            "output": {
                "addresses": [
                    "P-avax1tnuesf6cgwnjw7fxjyk7lhch0vhf0v95wj5jvy"
                ],
                "amount": 48944170378,
                "locktime": 0,
                "threshold": 1
            }
        }
    ],
    "rewardsOwner": {
        "addresses": [
            "P-avax19zfygxaf59stehzedhxjesads0p5jdvdfeedal0"
        ],
        "locktime": 0,
        "threshold": 1
    }
},
"credentials": [
{
    "signatures": [
        "0x6954e90b98437646fde0c1d54c12190fc23ae5e319c4d95dda56b53b4a23e43825251289cdc3728f1f1e0d48eac20e5c8f097baa9b49ea8a3cb6a41bb272d16"
    ]
},
{
    "signatures": [
        "0x6954e90b98437646fde0c1d54c12190fc23ae5e319c4d95dda56b53b4a23e43825251289cdc3728f1f1e0d48eac20e5c8f097baa9b49ea8a3cb6a41bb272d16"
    ]
},
{
    "id": "28KVjSw5h3XKGGuNpJXWY74EdnGg4TUWvCgEtJPymgQTvuudiugb"
},
"encoding": "json"
},
"id": 1
}

```

`platform.getTxStatus`

Gets a transaction's status by its ID. If the transaction was dropped, response will include a `reason` field with more information why the transaction was dropped.

Signature:

```
platform.getTxStatus({
    txID: string
}) -> {status: string}
```

`status` is one of:

- `Committed` : The transaction is (or will be) accepted by every node
- `Processing` : The transaction is being voted on by this node
- `Dropped` : The transaction will never be accepted by any node in the network, check `reason` field for more information
- `Unknown` : The transaction hasn't been seen by this node

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "platform.getTxStatus",
    "params": {
        "txID": "TAG9Ns1sa723mZy1GSogqWipK6Mvpaj7CAswVJGM6MkVJDF9Q"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "status": "Committed"
  },
  "id": 1
}
```

`platform.getUTXOs`

Gets the UTXOs that reference a given set of addresses.

Signature:

```
platform.getUTXOs(
  {
    addresses: []string,
    limit: int, // optional
    startIndex: { // optional
      address: string,
      utxo: string
    },
    sourceChain: string, // optional
    encoding: string, // optional
  },
) ->
{
  numFetched: int,
  utxos: []string,
  endIndex: {
    address: string,
    utxo: string
  },
  encoding: string,
}
```

- `utxos` is a list of UTXOs such that each UTXO references at least one address in `addresses`.
- At most `limit` UTXOs are returned. If `limit` is omitted or greater than 1024, it is set to 1024.
- This method supports pagination. `endIndex` denotes the last UTXO returned. To get the next set of UTXOs, use the value of `endIndex` as `startIndex` in the next call.
- If `startIndex` is omitted, will fetch all UTXOs up to `limit`.
- When using pagination (that is when `startIndex` is provided), UTXOs are not guaranteed to be unique across multiple calls. That is, a UTXO may appear in the result of the first call, and then again in the second call.
- When using pagination, consistency is not guaranteed across multiple calls. That is, the UTXO set of the addresses may have changed between calls.
- `encoding` specifies the format for the returned UTXOs. Can only be `hex` when a value is provided.

Example

Suppose we want all UTXOs that reference at least one of `P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5` and `P-avax1d09qn852zcy03sfc9hay21lmn9hsgnw4tp3dv6`.

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.getUTXOs",
  "params": {
    "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5", "P-avax1d09qn852zcy03sfc9hay21lmn9hsgnw4tp3dv6"],
    "limit": 5,
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

This gives response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "5",
    "utxos": [
      "0x0000a195046108a85e60f7a864bb567745a37f50c6af282103e47cc62f036cee40470000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000ae8b1b9444eed8de9a81b1222f00f1b4133330add23d8ac288bffa98b8527110000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000731ce04b1feefafa9f4291d869adc30a33463f315491e164d89be7d6d2d7890fc0000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978"
    ]
}
```

```

"0x00000b462030cc4734f24c0bc224cf0d16ee452ea6b67615517caffead123ab4fbf1500000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
"0x000054f6826c39bc957c0c6d44b70f961a994898999179cc32d21eb09c1908d7167b00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
],
{
  "endIndex": {
    "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "utxo": "kbUThAUfmBXUmRgTpG6r3nLj7rJUGho6xyht5nouNNypH45j"
  },
  "encoding": "hex"
},
{
  "id": 1
}

```

Since `numFetched` is the same as `limit`, we can tell that there may be more UTXOs that were not fetched. We call the method again, this time with `startIndex`:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.getUTXOs",
  "params": {
    "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "limit": 5,
    "startIndex": {
      "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "0x62fc816bb209857923770c286192ab1f9e3f1le4a7d4ba0943111c3bbfeb9e4a5ea72fae"
    },
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

This gives response:

```

{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "4",
    "utxos": [
      "0x000020e182dd51ee4cd31909fddd75bb3438d9431f8e4efce86a88a684f5c7fa09300000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
      "0x0000a71ba36c475c18eb65dc90f6e85c4fd4a462d51c5de3ac2cbddf47db4d99284e00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
      "0x0000925424f61cb13e0fbddecc66e1270de68de9667b85baa3fdc84741d048daa69fa00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
      "0x000082f30327514f819da6009fad92b5dba24d27db01e29ad7541aa8e6b6b554615c00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
    ],
    "endIndex": {
      "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "21jG2RfqyHUUgkTLe2tUp6ETGLriSDTW3th8JXFbPRNiSZ11jk"
    },
    "encoding": "hex"
  },
  "id": 1
}

```

Since `numFetched` is less than `limit`, we know that we are done fetching UTXOs and don't need to call this method again.

Suppose we want to fetch the UTXOs exported from the X Chain to the P Chain in order to build an ImportTx. Then we need to call `GetUTXOs` with the `sourceChain` argument in order to retrieve the atomic UTXOs:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.getUTXOs",
  "params": {
    "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "sourceChain": "X",
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

This gives response:

```
{
  "jsonrpc": "2.0",

```

```

"result": {
  "numFetched": "1",
  "utxos": [
    "0x00001f989ffaf18a18a59bdfbf209342aa61c6a62a67e8639d02bb3c8ddab315c6fa000000139c33a499ce4c33a3b09cdd2cfa01ae70dbf2d18b2d7d168524",
  ],
  "endIndex": {
    "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "utxo": "S5UKgWoVpoGFyxfisebmrmf8WqC7ZwcmYwS7XaDVZqoaFcCwK"
  },
  "encoding": "hex"
},
"id": 1
}

```

platform.getValidatorsAt

Get the validators and their weights of a Subnet or the Primary Network at a given P-Chain height.

Signature:

```

platform.getValidatorsAt(
{
  height: int,
  subnetID: string, // optional
}
)

```

- `height` is the P-Chain height to get the validator set at.
- `subnetID` is the Subnet ID to get the validator set of. If not given, gets validator set of the Primary Network.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.getValidatorsAt",
  "params": {
    "height":1
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

Example Response:

```

{
  "jsonrpc": "2.0",
  "result": {
    "validators": [
      "NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Ig": 2000000000000000,
      "NodeID-GWFcbFJZFFzreETSoWjPimr846mXEKCtu": 2000000000000000,
      "NodeID-MFrZFVCXPv5iCn6M9K6XduxGTYp891xZ": 2000000000000000,
      "NodeID-NFBbbJ4qCmNaCzeW7sxErhvWqvEQMnYCN": 2000000000000000,
      "NodeID-P7oB2McjBGgW2NXXWVYjV8JEDFoW9xDE5": 2000000000000000
    ]
  },
  "id": 1
}

```

platform.importAVAX

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Complete a transfer of AVAX from the X-Chain, or C-Chain to the P-Chain.

Before this method is called, you must call the X-Chain's [avm.export](#) or C-Chain's [avax.export](#) method with assetID `AVAX` to initiate the transfer.

Signature:

```

platform.importAVAX(
{
  to: string,
  sourceChain: string,
  from: []string, // optional
  changeAddr: string, // optional
  username: string,
  password: string
}
) ->
{
  tx: string,
  changeAddr: string
}

```

- `to` is the ID of the address the AVAX is imported to. This must be the same as the `to` argument in the corresponding call to the X-Chain's or C-Chain's `export`.
- `sourceChain` is the chain the funds are coming from. Must be either `"x"` or `"c"`.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user that controls from and change addresses.
- `password` is `username`'s password.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.importAVAX",
  "params": {
    "to": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "from": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u",
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P

```

Example Response:

```

{
  "jsonrpc": "2.0",
  "result": {
    "txID": "P63NjowXaQJXt5cmsspqdoD3VcuQdxUPM5eoZE2Vcg63aVEx8R",
    "changeAddr": "P-avax103y30cxeulkjfe3kwfnpt432ylmnux8r73r8u"
  },
  "id": 1
}

```

`platform.importKey`

::caution

Deprecated as of [v1.9.12](#).

:::

:::warning

Not recommended for use on Mainnet. See warning notice in [Keystore API](#).

:::

Give a user control over an address by providing the private key that controls the address.

Signature:

```

platform.importKey({
  username: string,
  password: string,
  privateKey:string
}) -> {address: string}

```

- Add `privateKey` to `username`'s set of private keys. `address` is the address `username` now controls with the private key.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "platform.importKey",
    "params": {
        "username": "myUsername",
        "password": "myPassword",
        "privateKey": "PrivateKey-2w4XiXxPfQK4TypYqnohRL8DRNTz9cGiGmWQlzmgEqD9c9KWlQ"
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "address": "P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"  
    }  
}
```

platform.issueTx

Issue a transaction to the Platform Chain.

Signature:

```
platform.issueTx({
  tx: string,
  encoding: string, // optional
}) -> {txID: string}
```

- `tx` is the byte representation of a transaction.
 - `encoding` specifies the encoding format for the transaction bytes. Can only be `hex` when a value is provided.
 - `txID` is the transaction's ID.

Example Call:

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "result": {  
        "txID": "G3BuH6ytQ2averrLxJJugjWZHTrubzCrUZEhoheG5JMqL5ccY"  
    },  
    "id": 1  
}
```

platform.listAddresses

Conclusion

Deprecated as of v1.9.12

111

...warning

Not recommended for use on Meijnet. See warning notice in Keynotes API.

List addresses controlled by the given user

Signature:

```
platform.listAddresses({
  username: string,
  password: string
}) -> {addresses: []string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "platform.listAddresses",
  "params": {
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "addresses": ["P-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"]
  },
  "id": 1
}
```

platform.sampleValidators

Sample validators from the specified Subnet.

Signature:

```
platform.sampleValidators(
  {
    size: int,
    subnetID: string, // optional
  }
) ->
{
  validators: []string
}
```

- `size` is the number of validators to sample.
- `subnetID` is the Subnet to sampled from. If omitted, defaults to the Primary Network.
- Each element of `validators` is the ID of a validator.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "platform.sampleValidators",
  "params": {
    "size": 2
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "validators": [
      "NodeID-MFrZFVCXPv5iCn6M9K6XduxGTYp891xxZ",
      "NodeID-NFBbbJ4qCmNaCzeW7sxErhvWqvEQMnYcN"
    ]
  }
}
```

platform.validatedBy

Get the Subnet that validates a given blockchain.

Signature:

```
platform.validatedBy(  
  {  
    blockchainID: string  
  }  
) -> {subnetID: string}
```

- `blockchainID` is the blockchain's ID.
- `subnetID` is the ID of the Subnet that validates the blockchain.

Example Call:

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "method": "platform.validatedBy",  
  "params": {  
    "blockchainID": "KDYHHKjM4yTJTT8H8qPs5KXzE6gQH5TZrmPlqVr1P6qECj3XN"  
  },  
  "id": 1  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "subnetID": "2bRCr6B4MiEfSjidDwxDpdCyviwnfUVqB2HGwhm947w9YYqb7r"  
  },  
  "id": 1  
}
```

platform.validates

Get the IDs of the blockchains a Subnet validates.

Signature:

```
platform.validates(  
  {  
    subnetID: string  
  }  
) -> {blockchainIDs: []string}
```

- `subnetID` is the Subnet's ID.
- Each element of `blockchainIDs` is the ID of a blockchain the Subnet validates.

Example Call:

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "method": "platform.validates",  
  "params": {  
    "subnetID": "2bRCr6B4MiEfSjidDwxDpdCyviwnfUVqB2HGwhm947w9YYqb7r"  
  },  
  "id": 1  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/P
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "blockchainIDs": [  
      "KDYHHKjM4yTJTT8H8qPs5KXzE6gQH5TZrmPlqVr1P6qECj3XN",  
      "2tHFqEAAJ6b33dromYMqfgavGPF3iCpdG3hwNMiart2aB5QHi"  
    ]  
  },  
  "id": 1  
}
```

sidebar_position: 14

Subnet-EVM API

[Subnet-EVM](#) APIs are identical to [Coreth](#) APIs, except Avalanche Specific APIs starting with `avax`. Subnet-EVM also supports standard Ethereum APIs as well. For more information about Coreth APIs see [here](#).

Subnet-EVM has some additional APIs that are not available in Coreth.

`eth_feeConfig`

Subnet-EVM comes with an API request for getting fee config at a specific block. You can use this API to check your activated fee config.

Signature:

```
eth_feeConfig([blk BlkNrOrHash]) -> {feeConfig: json}
```

- `blk` is the block number or hash at which to retrieve the fee config. Defaults to the latest block if omitted.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "eth_feeConfig",
  "params": [
    "latest"
  ],
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qegeh6tt/rpc
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "feeConfig": {
      "gasLimit": 1500000,
      "targetBlockRate": 2,
      "minBaseFee": 33000000000,
      "targetGas": 1500000,
      "baseFeeChangeDenominator": 36,
      "minBlockGasCost": 0,
      "maxBlockGasCost": 1000000,
      "blockGasCostStep": 200000
    },
    "lastChangedAt": 0
  }
}
```

`eth_getChainConfig`

`eth_getChainConfig` returns the Chain Config of the blockchain. This API is enabled by default with `internal-blockchain` namespace.

This API exists on the C-Chain as well, but in addition to the normal Chain Config returned by the C-Chain `eth_getChainConfig` on `subnet-evm` additionally returns the upgrade config, which specifies network upgrades activated after the genesis. **Signature:**

```
eth_getChainConfig() -> {chainConfig: json}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "eth_getChainConfig",
  "params": []
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/Nvqcm33CX2XABS62izsAcVUkavfnzp1Sc5k413wn5Nrf7Qjt7/rpc
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "chainId": 43214,
    "feeConfig": {
```

```

    "gasLimit": 8000000,
    "targetBlockRate": 2,
    "minBaseFee": 33000000000,
    "targetGas": 15000000,
    "baseFeeChangeDenominator": 36,
    "minBlockGasCost": 0,
    "maxBlockGasCost": 1000000,
    "blockGasCostStep": 200000
  },
  "allowFeeRecipients": true,
  "homesteadBlock": 0,
  "eip150Block": 0,
  "eip150Hash": "0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",
  "eip155Block": 0,
  "eip158Block": 0,
  "byzantiumBlock": 0,
  "constantinopleBlock": 0,
  "petersburgBlock": 0,
  "istanbulBlock": 0,
  "muirGlacierBlock": 0,
  "subnetEVMTimestamp": 0,
  "contractDeployerAllowListConfig": {
    "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
    "blockTimestamp": 0
  },
  "contractNativeMinterConfig": {
    "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
    "blockTimestamp": 0
  },
  "feeManagerConfig": {
    "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
    "blockTimestamp": 0
  },
  "upgrades": {
    "precompileUpgrades": [
      {
        "feeManagerConfig": {
          "adminAddresses": null,
          "blockTimestamp": 1661541259,
          "disable": true
        }
      },
      {
        "feeManagerConfig": {
          "adminAddresses": null,
          "blockTimestamp": 1661541269
        }
      }
    ]
  }
}
}

```

eth_getActivePrecompilesAt

`eth_getActivePrecompilesAt` returns activated precompiles at a specific timestamp. If no timestamp is provided it returns the latest block timestamp. This API is enabled by default with `internal-blockchain` namespace.

Signature:

```
eth_getActivePrecompilesAt([timestamp uint]) -> {precompiles: []Precompile}
```

- `timestamp` specifies the timestamp to show the precompiles active at this time. If omitted it shows precompiles activated at the latest block timestamp.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "eth_getActivePrecompilesAt",
  "params": [],
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/Nvqcm33CX2XABS62iZsAcVUkavfnzp1Sc5k413wn5Nrf7Qjt7/rpc
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "contractDeployerAllowListConfig": {
      "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
      "blockTimestamp": 0
    },
    "contractNativeMinterConfig": {
      "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
      "blockTimestamp": 0
    },
    "feeManagerConfig": {
      "adminAddresses": ["0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc"],
      "blockTimestamp": 0
    }
  }
}
```

description: The X-Chain is an instance of the Avalanche Virtual Machine (AVM) **sidebar_position:** 5

Exchange Chain (X-Chain) API

The [X-Chain](#), Avalanche's native platform for creating and trading assets, is an instance of the Avalanche Virtual Machine (AVM). This API allows clients to create and trade assets on the X-Chain and other instances of the AVM.

Format

This API uses the `json 2.0` RPC format. For more information on making JSON RPC calls, see [here](#).

Endpoints

`/ext/bc/x` to interact with the X-Chain.

`/ext/bc/blockchainID` to interact with other AVM instances, where `blockchainID` is the ID of a blockchain running the AVM.

Methods

`avm.buildGenesis`

Given a JSON representation of this Virtual Machine's genesis state, create the byte representation of that state.

Endpoint

This call is made to the AVM's static API endpoint:

`/ext/vm/avm`

Note: addresses should not include a chain prefix (that is `x-`) in calls to the static API endpoint because these prefixes refer to a specific chain.

Signature:

```
avm.buildGenesis({
  networkID: int,
  genesisData: JSON,
  encoding: string, //optional
}) -> {
  bytes: string,
  encoding: string,
}
```

Encoding specifies the encoding format to use for arbitrary bytes, that is the genesis bytes that are returned. Can only be `hex` when a value is provided.

`genesisData` has this form:

```
{
  "genesisData" :
  {
    "assetAlias1": {           // Each object defines an asset
      "name": "human readable name",
      "symbol": "AVAL",        // Symbol is between 0 and 4 characters
      "initialState": {
        "fixedCap" : [          // Choose the asset type.
          {                   // Can be "fixedCap", "variableCap", "limitedTransfer", "nonFungible"
            "amount":1000, // At genesis, address A has
            "address": "A" // 1000 units of asset
          }
        ]
      }
    }
  }
}
```

```

        },
        {
            "amount":5000, // At genesis, address B has
            "address":"B" // 1000 units of asset
        },
        ... // Can have many initial holders
    ]
}
},
"assetAliasCanBeAnythingUnique": { // Asset alias can be used in place of assetID in calls
    "name": "human readable name", // names need not be unique
    "symbol": "AVAL", // symbols need not be unique
    "initialState": {
        "variableCap" : [ // No units of the asset exist at genesis
            {
                "minters": [ // The signature of A or B can mint more of
                    "A", // the asset.
                    "B"
                ],
                "threshold":1
            },
            {
                "minters": [ // The signatures of 2 of A, B and C can mint
                    "A", // more of the asset
                    "B",
                    "C"
                ],
                "threshold":2
            },
            ...
        ] // Can have many minter sets
    }
},
...
// Can list more assets
}
}
}

```

Example Call:

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "id" : 1,
    "method" : "avm.buildGenesis",
    "params" : {
        "networkId": 16,
        "genesisData": {
            "asset1": {
                "name": "myFixedCapAsset",
                "symbol": "MFCA",
                "initialState": {
                    "fixedCap" : [
                        {
                            "amount":100000,
                            "address": "avax13ery2kvdrkd2nkquvs892g18hg7mq4a6ufnra6"
                        },
                        {
                            "amount":100000,
                            "address": "avax1rvks3vpe4cm9yc0rrk8d5855nd6yxxutfc2h2r"
                        },
                        {
                            "amount":50000,
                            "address": "avax1ntj922dj4crc4pre4e0xt3dyj0t5rsw9uw0tus"
                        },
                        {
                            "amount":50000,
                            "address": "avax1lyk0xzmqyyaxn26sqceuky2tc2fh2q327vcwvda"
                        }
                    ]
                }
            },
            "asset2": {
                "name": "myVarCapAsset",
                "symbol": "MVCA",
                "initialState": {
                    "variableCap" : [
                        {
                            "minters": [

```

```
        "avax1xkcfg6avc94ct3qh2mtdg47thsk8nrflnrgwjqqr",
        "avax14e2s22wxvfc3c7309txxpqsoqe9tjwwtk0dme8e"
    ],
    "threshold":1
},
{
    "minters": [
        "avax1y8pvney82gjyqr7kqzp72pqym6xlch9gt5grck",
        "avax1c5cm0gem70rd8dcnpel63apzfnfxye9kd4wwe",
        "avax12euam2lwtwa8apvfdl700ckhg86euag2hlhmyw"
    ],
    "threshold":2
}
]
}
}
},
"encoding": "hex"
}
)' -H 'content-type:application/json;' 127.0.0.1:9650/ext/vm/avm
```

Example Response:

`avm.createAddress`

...:caution

Deprecated as of [v1.9.12](#).

•

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Create a new address controlled by the given user.

Signature:

```
    avm.createAddress ({  
        username: string,  
        password: string  
    }) -> {address: string}
```

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "avm.createAddress",
    "params": {
        "username": "myUsername",
        "password": "myPassword"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
  },
  "id": 1
}
```

```
avm.createFixedCapAsset
```

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Create a new fixed-cap, fungible asset. A quantity of it is created at initialization and then no more is ever created. The asset can be sent with `avm.send`.

Signature:

```
avm.createFixedCapAsset({
  name: string,
  symbol: string,
  denomination: int, //optional
  initialHolders: []{
    address: string,
    amount: int
  },
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) ->
{
  assetID: string,
  changeAddr: string
}
```

- `name` is a human-readable name for the asset. Not necessarily unique.
- `symbol` is a shorthand symbol for the asset. Between 0 and 4 characters. Not necessarily unique. May be omitted.
- `denomination` determines how balances of this asset are displayed by user interfaces. If `denomination` is 0, 100 units of this asset are displayed as 100. If `denomination` is 1, 100 units of this asset are displayed as 10.0. If `denomination` is 2, 100 units of this asset are displayed as 1.00, etc. Defaults to 0.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` and `password` denote the user paying the transaction fee.
- Each element in `initialHolders` specifies that `address` holds `amount` units of the asset at genesis.
- `assetID` is the ID of the new asset.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.createFixedCapAsset",
  "params": {
    "name": "myFixedCapAsset",
    "symbol": "MFCA",
    "initialHolders": [
      {
        "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
        "amount": 10000
      },
      {
        "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
        "amount": 50000
      }
    ],
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "assetID": "ziKfqRXCZgHLgZ4rxGU9Qbycdzuq5DRY4tdSNS9ku8kcNxNLD",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

`avm.createNFTAsset`

::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Create a new non-fungible asset. No units of the asset exist at initialization. Minters can mint units of this asset using `avm.mintNFT`.

Signature:

```
avm.createNFTAsset({
  name: string,
  symbol: string,
  minterSets: []{
    minters: []string,
    threshold: int
  },
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) ->
{
  assetID: string,
  changeAddr: string,
}
```

- `name` is a human-readable name for the asset. Not necessarily unique.
- `symbol` is a shorthand symbol for the asset. Between 0 and 4 characters. Not necessarily unique. May be omitted.
- `minterSets` is a list where each element specifies that `threshold` of the addresses in `minters` may together mint more of the asset by signing a minting transaction.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` pays the transaction fee.
- `assetID` is the ID of the new asset.
- `changeAddr` in the result is the address where any change was sent.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" :"avm.createNFTAsset",
  "params" :{
    "name": "Coincert",
    "symbol": "TIXX",
    "minterSets": [
      {
        "minters": [
          "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
        ],
        "threshold": 1
      }
    ],
    "from": ["X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "assetID": "2KGdt2HpFKpTH5CtGZjYt5XPWs6Pv9DLoRBhiFfntbezdRvZWP",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  },
  "id": 1
}
```

`avm.createVariableCapAsset`

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Create a new variable-cap, fungible asset. No units of the asset exist at initialization. Minters can mint units of this asset using `avm.mint`.

Signature:

```
avm.createVariableCapAsset({
  name: string,
  symbol: string,
  denomination: int, //optional
  minterSets: []{
    minters: []string,
    threshold: int
  },
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) ->
{
  assetID: string,
  changeAddr: string,
}
```

- `name` is a human-readable name for the asset. Not necessarily unique.
- `symbol` is a shorthand symbol for the asset. Between 0 and 4 characters. Not necessarily unique. May be omitted.
- `denomination` determines how balances of this asset are displayed by user interfaces. If denomination is 0, 100 units of this asset are displayed as 100. If denomination is 1, 100 units of this asset are displayed as 10.0. If denomination is 2, 100 units of this asset are displayed as .100, etc.
- `minterSets` is a list where each element specifies that `threshold` of the addresses in `minters` may together mint more of the asset by signing a minting transaction.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` pays the transaction fee.
- `assetID` is the ID of the new asset.
- `changeAddr` in the result is the address where any change was sent.

Example Call:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "id" : 1,
  "method" :"avm.createVariableCapAsset",
  "params" :{
    "name":"myVariableCapAsset",
    "symbol":"MFCA",
    "minterSets": [
      {
        "minters":[
          "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"
        ],
        "threshold": 1
      },
      {
        "minters": [
          "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
          "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
          "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"
        ],
        "threshold": 2
      }
    ],
    "from":["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr":"X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username":"myUsername",
    "password":"myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "assetID": "2QbzFE7J4MAny9iXHUwq8Pz8SpFhWk3maCw4SkinVPv6wPmAbK",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

avm.export

::caution

Deprecated as of [v1.9.12](#).

:::

:: Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Send an asset from the X-Chain to the P-Chain or C-Chain. After calling this method, you must call the [C-Chain's avax.import](#) or the [P-Chain's platform.importAVAX](#) to complete the transfer.

Signature:

```
avm.export({
  to: string,
  amount: int,
  assetID: string,
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string,
}) ->
{
  txID: string,
  changeAddr: string,
}
```

- `to` is the P-Chain or C-Chain address the asset is sent to.
- `amount` is the amount of the asset to send.
- `assetID` is the asset id of the asset which is sent. Use `AVAX` for AVAX exports.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- The asset is sent from addresses controlled by `username`
- `password` is `username`'s password.
- `txID` is this transaction's ID.
- `changeAddr` in the result is the address where any change was sent.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.export",
  "params": {
    "to": "C-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "amount": 10,
    "assetID": "AVAX",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "2Eu16yNaepP57XrrJgjKGpiEDandpiGWW8xbUm6wcTYny3fejj",
```

```
        "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
    },
    "id": 1
}
```

avm.exportKey

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Get the private key that controls a given address. The returned private key can be added to a user with [avm.importKey](#).

Signature:

```
avm.exportKey({
  username: string,
  password: string,
  address: string
}) -> {privateKey: string}
```

- `username` must control `address`.
- `privateKey` is the string representation of the private key that controls `address`.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.exportKey",
  "params": {
    "username": "myUsername",
    "password": "myPassword",
    "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "privateKey": "PrivateKey-2w4XiXxPfQK4TypYqnohRL8DRNTz9cGiGmwQ1zmgEqD9c9KWLq"
  }
}
```

avm.getAddressTxs

:::caution

Deprecated as of [v1.9.12](#).

:::

Returns all transactions that change the balance of the given address. A transaction is said to change an address's balance if either is true:

- A UTXO that the transaction consumes was at least partially owned by the address.
- A UTXO that the transaction produces is at least partially owned by the address.

:::tip Note: Indexing (`index-transactions`) must be enabled in the X-chain config. :::

Signature:

```
avm.getAddressTxs({
  address: string,
  cursor: uint64,      // optional, leave empty to get the first page
  assetID: string,
  pageSize: uint64    // optional, defaults to 1024
}) -> {
  txIDs: []string,
  cursor: uint64,
}
```

Request Parameters:

- `address` : The address for which we're fetching related transactions
- `assetID` : Only return transactions that changed the balance of this asset. Must be an ID or an alias for an asset.
- `pageSize` : Number of items to return per page. Optional. Defaults to 1024.

Response Parameter:

- `txIDs` : List of transaction IDs that affected the balance of this address.
- `cursor` : Page number or offset. Use this in request to get the next page.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" : "avm.getAddressTxs",
  "params" : {
    "address": "X-local1kpprmfpzzm51xyene32f6lr7j0aj7gxsu6hp9y",
    "assetID": "AVAX",
    "pageSize": 20
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txIDs": ["SsJF7KKwxUJkczygwmgLqo3XVRotmpKP8rMp74cpLuNLfwf6"],
    "cursor": "1"
  },
  "id": 1
}
```

avm.getAllBalances

::caution

Deprecated as of [v1.9.12](#).

:::

Get the balances of all assets controlled by a given address.

Signature:

```
avm.getAllBalances({address:string}) -> {
  balances: []{
    asset: string,
    balance: int
  }
}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" : "avm.getAllBalances",
  "params" : {
    "address": "X-avax1c79e0dd0suspl7dc8udq34jgk2yvve7hapvdyht"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "balances": [
      {
        "asset": "AVAX",
        "balance": "102"
      },
      {
        "asset": "2sdnziCz37Jov3QSNMXcFRGFJ1tgauaj6L7qfk7yUcRPFQMC79",
        "balance": "10000"
      }
    ]
  }
}
```

```
        ],
    },
    "id": 1
}
```

avm.getAssetDescription

Get information about an asset.

Signature:

```
avm.getAssetDescription({assetID: string}) -> {
  assetId: string,
  name: string,
  symbol: string,
  denomination: int
}
```

- `assetID` is the id of the asset for which the information is requested.
- `name` is the asset's human-readable, not necessarily unique name.
- `symbol` is the asset's symbol.
- `denomination` determines how balances of this asset are displayed by user interfaces. If denomination is 0, 100 units of this asset are displayed as 100. If denomination is 1, 100 units of this asset are displayed as 10.0. If denomination is 2, 100 units of this asset are displays as .100, etc.

:::note

The AssetID for AVAX differs depending on the network you are on.

Mainnet: FvwEAhmxKfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z

Testnet: U8iRqJoiJm8xZHAacmvYyZVwqQx6uDNTQeP3CQ6fcgQk3JqnK

For finding the `assetID` of other assets, this [info] might be useful. Also, `avm.getUTXOs` returns the `assetID` in its output.

:::

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getAssetDescription",
  "params": {
    "assetID": "FvwEAhmxKfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/x
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "assetID": "FvwEAhmxKfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
    "name": "Avalanche",
    "symbol": "AVAX",
    "denomination": "9"
  },
  "id": 1
}
```

avm.getBalance

:::caution

Deprecated as of [v1.9.12](#).

:::

Get the balance of an asset controlled by a given address.

Signature:

```
avm.getBalance({
  address: string,
  assetID: string
}) -> {balance: int}
```

- `address` owner of the asset
- `assetID` id of the asset for which the balance is requested

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getBalance",
  "params": {
    "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "assetID": "2pYGetDWyKdHxpFjh2LHeoLNCH6H5vxxCx#QtFnnFaYxLsgtHC"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "balance": "29999999999900",
    "utxoIDs": [
      {
        "txID": "WPQdyLNqHfiEKp4zcCpayRHYDVYuh1hqs9c1RggZXS4VPgdvo",
        "outputIndex": 1
      }
    ]
  }
}
```

avm.getBlock

Returns the block with the given id.

Signature:

```
avm.getBlock({
  blockID: string
  encoding: string // optional
}) -> {
  block: string,
  encoding: string
}
```

Request:

- `blockID` is the block ID. It should be in cb58 format.
- `encoding` is the encoding format to use. Can be either `hex` or `json`. Defaults to `hex`.

Response:

- `block` is the transaction encoded to `encoding`.
- `encoding` is the `encoding`.

Hex Example**Example Call:**

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "avm.getBlock",
  "params": {
    "blockID": "tXJ4xwmR8soHE6DzRNMQPtiwQvuYsHn6eLLBzo2moDqBquqy6",
    "encoding": "hex"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "block": "0x0000000000200000000641ad33ede17f652512193721df87994f783ec806bb5640c39ee73676caffcc3215e065100000000049a80a0000001000000e000(0
    "encoding": "hex"
  },
  "id": 1
}
```

`avm.getBlockByHeight`

Returns block at the given height.

Signature:

```
avm.getBlockByHeight({
  height: string
  encoding: string // optional
}) -> {
  block: string,
  encoding: string
}
```

Request:

- `blockHeight` is the block height. It should be in `string` format.
 - `encoding` is the encoding format to use. Can be either `hex` or `json`. Defaults to `hex`.

Response:

- `block` is the transaction encoded to `encoding`.
 - `encoding` is the `encoding`.

Hex Example

Example Cells

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "avm.getBlockByHeight",
    "params": {
        "height": "275686313486",
        "encoding": "hex"
    },
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

`avm_getHeight`

Returns the height of the last accepted block

Signature:

```
avm.getHeight() ->  
{  
    height: uint64  
}
```

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "method": "avm.getHeight",
    "params": {},
    "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "height": "5094088"  
  },  
}
```

```
        "id": 1
    }
```

avm.getTx

Returns the specified transaction. The `encoding` parameter sets the format of the returned transaction. Can be either `"hex"` or `"json"`. Defaults to `"hex"`.

Signature:

```
avm.getTx({
  txID: string,
  encoding: string, //optional
}) -> {
  tx: string,
  encoding: string,
}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getTx",
  "params": {
    "txID": "2oJCbb8pfpxEHAf9A8CdN4Afj9VSR3xxyzNkf8tDv7aM1sfNFL",
    "encoding": "json"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "tx": {
      "unsignedTx": {
        "networkID": 1,
        "blockchainID": "2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM",
        "outputs": [],
        "inputs": [
          {
            "txID": "2jbZUvi6nHy3Pgmk8xcMpSg5cW6epkPqdKkHSCweb4eRxtq4k9",
            "outputIndex": 1,
            "assetID": "FvwEAhmxAfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
            "fxID": "spdxUxVJQbx85MGxMHbKwlsHxMnSqJ3QBzDyDYEPM3h6TLuxqQ",
            "input": {
              "amount": 2570382395,
              "signatureIndices": [
                0
              ]
            }
          }
        ],
        "memo": "0x",
        "destinationChain": "11111111111111111111111111111111LpoYY",
        "exportedOutputs": [
          {
            "assetID": "FvwEAhmxAfeiG8SnEvq42hc6whRyY3EFYAvebMqDNDGCgxN5Z",
            "fxID": "spdxUxVJQbx85MGxMHbKwlsHxMnSqJ3QBzDyDYEPM3h6TLuxqQ",
            "output": {
              "addresses": [
                "X-avax1tnuesf6cqwnjw7fxjyk7lhch0vhf0v95wj5jvy"
              ],
              "amount": 2569382395,
              "locktime": 0,
              "threshold": 1
            }
          }
        ]
      },
      "credentials": [
        {
          "fxID": "spdxUxVJQbx85MGxMHbKwlsHxMnSqJ3QBzDyDYEPM3h6TLuxqQ",
          "credential": {
            "signatures": [

```

`"0x46ebcbcfbee3ece1fd15015204045cf3cb77f42c48d0201fc150341f91f086f177cfca8894ca9b4a0c55d6950218e4ea8c01d5c4aefb85cd7264b47bd57d2244`

```

        ]
    }
}
],
"id": "2oJCbb8pfpxEHAF9A8CdN4Afj9VSR3xyzNkf8tDv7aM1sfNFL"
},
"encoding": "json"
},
"id": 1
}

```

Where:

- `credentials` is a list of this transaction's credentials. Each credential proves that this transaction's creator is allowed to consume one of this transaction's inputs. Each credential is a list of signatures.
- `unsignedTx` is the non-signature portion of the transaction.
- `networkID` is the ID of the network this transaction happened on. (Avalanche Mainnet is `1`.)
- `blockchainID` is the ID of the blockchain this transaction happened on. (Avalanche Mainnet X-Chain is `2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM`.)
- Each element of `outputs` is an output (UTXO) of this transaction that is not being exported to another chain.
- Each element of `inputs` is an input of this transaction which has not been imported from another chain.
- Import Transactions have additional fields `sourceChain` and `importedInputs`, which specify the blockchain ID that assets are being imported from, and the inputs that are being imported.
- Export Transactions have additional fields `destinationChain` and `exportedOutputs`, which specify the blockchain ID that assets are being exported to, and the UTXOs that are being exported.

An output contains:

- `assetID` : The ID of the asset being transferred. (The Mainnet Avax ID is `FvwEAhmxKfeiG8SnEvq42hc6whRyY3EFYAvabMqDNDGCgxN5Z`.)
- `fxID` : The ID of the FX this output uses.
- `output` : The FX-specific contents of this output.

Most outputs use the secp256k1 FX, look like this:

```
{
  "assetID": "FvwEAhmxKfeiG8SnEvq42hc6whRyY3EFYAvabMqDNDGCgxN5Z",
  "fxID": "spdxUxVJQbx85MGxMhbKwlsHxMnSgJ3QBzDyDYEPh6TLuxqQ",
  "output": {
    "addresses": ["X-avax126rd3w35xwkmj8670zvf7y5r8k36qa9z9803wm"],
    "amount": 1500084210,
    "locktime": 0,
    "threshold": 1
  }
}
```

The above output can be consumed after Unix time `locktime` by a transaction that has signatures from `threshold` of the addresses in `addresses`.

avm.getTxStatus

:::caution Deprecated as of v1.10.0. :::

Get the status of a transaction sent to the network.

Signature:

```
avm.getTxStatus({txID: string}) -> {status: string}
```

`status` is one of:

- `Accepted` : The transaction is (or will be) accepted by every node
- `Processing` : The transaction is being voted on by this node
- `Rejected` : The transaction will never be accepted by any node in the network
- `Unknown` : The transaction hasn't been seen by this node

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getTxStatus",
  "params": {
    "txID": "2QouvFWUbjuySRxeX5xMbNCuAaKWfbk5FeEa2JmoF85RKLk2dD"
  }
}' -H 'content-type:application/json,' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "status": "Accepted"
  }
}
```

avm.getUTXOs

Gets the UTXOs that reference a given address. If `sourceChain` is specified, then it will retrieve the atomic UTXOs exported from that chain to the X Chain.

Signature:

```
avm.getUTXOs({
  addresses: []string,
  limit: int, //optional
  startIndex: { //optional
    address: string,
    utxo: string
  },
  sourceChain: string, //optional
  encoding: string //optional
}) -> {
  numFetched: int,
  utxos: []string,
  endIndex: {
    address: string,
    utxo: string
  },
  sourceChain: string, //optional
  encoding: string
}
```

- `utxos` is a list of UTXOs such that each UTXO references at least one address in `addresses`.
- At most `limit` UTXOs are returned. If `limit` is omitted or greater than 1024, it is set to 1024.
- This method supports pagination. `endIndex` denotes the last UTXO returned. To get the next set of UTXOs, use the value of `endIndex` as `startIndex` in the next call.
- If `startIndex` is omitted, will fetch all UTXOs up to `limit`.
- When using pagination (when `startIndex` is provided), UTXOs are not guaranteed to be unique across multiple calls. That is, a UTXO may appear in the result of the first call, and then again in the second call.
- When using pagination, consistency is not guaranteed across multiple calls. That is, the UTXO set of the addresses may have changed between calls.
- `encoding` sets the format for the returned UTXOs. Can only be `hex` when a value is provided.

Example

Suppose we want all UTXOs that reference at least one of `X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5` and `X-avax1d09qn852zcy03sfc9hay21lmn9hsgnw4tp3dv6`.

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getUTXOs",
  "params": {
    "addresses": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5", "X-avax1d09qn852zcy03sfc9hay21lmn9hsgnw4tp3dv6"],
    "limit": 5,
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

This gives response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "5",
    "utxos": [
      "0x0000a195046108a85e60f7a864bb567745a37f50c6af282103e47cc62f036cee404700000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000ae8b1b94444eed8de9a81b1222f00f1b4133330add23d8ac288bffa98b85271100000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000731ce04b1feef9f4291d869adc30a33463f315491e164d89be7d6d2d7890fc00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000b462030cc4734f24c0bc224cf0d16ee452ea6b67615517caffead123ab4fbff1500000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978",
      "0x0000b462030cc4734f24c0bc224cf0d16ee452ea6b67615517caffead123ab4fbff1500000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978"
    ]
  }
}
```

```

"0x0000054f6826c39bc957c0c6d44b70f961a994898999179cc32d21eb09c1908d7167b00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
],
"endIndex": {
  "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
  "utxo": "kbUThAUfmBXUmRgTpqD6r3nLj7rJUGho6xyht5nouNNypH45j"
},
"encoding": "hex"
},
"id": 1
}

```

Since `numFetched` is the same as `limit`, we can tell that there may be more UTXOs that were not fetched. We call the method again, this time with `startIndex`:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "avm.getUTXOs",
  "params": {
    "addresses": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "limit": 5,
    "startIndex": {
      "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "kbUThAUfmBXUmRgTpqD6r3nLj7rJUGho6xyht5nouNNypH45j"
    },
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X

```

This gives response:

```

{
  "jsonrpc": "2.0",
  "result": {
    "numFetched": "4",
    "utxos": [
      "0x000020e182dd51ee4cd31909fddd75bb3438d9431f8e4efce86a88a684f5c7fa09300000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786",
      "0x0000a71ba36c475c18eb65dc90f6e85c4fd4a462d51c5de3ac2cbddf47db4d99284e00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786",
      "0x0000925424f61cb13e0fbdecc66e1270de68de9667b85baa3fdc84741d048daa69fa00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786",
      "0x000082f30327514f819da6009fad92b5dba24d27db01e29ad7541aa8e6b6b554615c00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a779786
    ],
    "endIndex": {
      "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
      "utxo": "21jG2RfgyHUUgkTLE2tUp6ETGLriSDTW3th8JXFbPRNISZ11jK"
    },
    "encoding": "hex"
  },
  "id": 1
}

```

Since `numFetched` is less than `limit`, we know that we are done fetching UTXOs and don't need to call this method again.

Suppose we want to fetch the UTXOs exported from the P Chain to the X Chain in order to build an ImportTx. Then we need to call `GetUTXOs` with the `sourceChain` argument in order to retrieve the atomic UTXOs:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.getUTXOs",
  "params": {
    "addresses": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5", "X-avax1d09qn852zcy03sf9hay21lmn9hsgnw4tp3dv6"],
    "limit": 5,
    "sourceChain": "P",
    "encoding": "hex"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X

```

This gives response:

```
{
  "jsonrpc": "2.0",
  "result": {

```

```

    "numFetched": "1",
    "utxos": [
      {
        "txid": "0x000001f989ffaf18a18a59bdfbf209342aa61c6a62a67e8639d02bb3c8ddab315c6fa0000000039c33a499ce4c33a3b09cdd2cfa01ae70dbf2d18b2d7d168524",
        "index": 1,
        "txIndex": 1,
        "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
        "utxo": "2Sz2XwRYqUHwPeiKoRnZ6ht88YqzAF1SQjMYZQQaB5wBFkAqST"
      }
    ],
    "encoding": "hex"
  },
  "id": 1
}

```

avm.import

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Finalize a transfer of an asset from the P-Chain or C-Chain to the X-Chain. Before this method is called, you must call the P-Chain's [platform.exportAVAX](#) or C-Chain's [avax.export](#) method to initiate the transfer.

Signature:

```

avm.import({
  to: string,
  sourceChain: string,
  username: string,
  password: string,
}) -> {txID: string}

```

- `to` is the address the AVAX is sent to. This must be the same as the `to` argument in the corresponding call to the P-Chain's `exportAVAX` or C-Chain's `export`.
- `sourceChain` is the ID or alias of the chain the AVAX is being imported from. To import funds from the C-Chain, use "C".
- `username` is the user that controls `to`.
- `txID` is the ID of the newly created atomic transaction.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "aevm.import",
  "params": {
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "sourceChain": "C",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "2gXpf4jFoMAWQ3rxBfavgFFSdLkL2eFUYprKsUQuEdB5H6Jo1H"
  },
  "id": 1
}
```

aevm.importKey

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Give a user control over an address by providing the private key that controls the address.

Signature:

```
    username: string,  
    password: string,  
    privateKey: string  
}) -> {address: string}
```

- Add `privateKey` to `username`'s set of private keys. `address` is the address `username` now controls with the private key.

Example Call:

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id" : 1,
    "method" : "avm.importKey",
    "params" : {
        "username": "myUsername",
        "password": "myPassword",
        "privateKey": "PrivateKey-2w4XiXxPfQK4TypYqnohRL8DRNTz9cGiGmwQlzmgEqD9c9KWLq
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{  
    "jsonrpc": "2.0",  
    "id": 1,  
    "result": {  
        "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"  
    }  
}
```

avm.issueTx

Send a signed transaction to the network. `encoding` specifies the format of the signed transaction. Can only be `hex` when a value is provided.

Signature:

```
    avm.issueTx({
      tx: string,
      encoding: string, //optional
    }) -> {
      txID: string
    }
  }
```

Example Call:

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "getinfo",
  "params": {}
}
```

avm.listAddresses

...caution

Deprecated as of v1.9.12.

* * *

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

List addresses controlled by the given user.

Signature:

```
avm.listAddresses({
  username: string,
  password: string
}) -> {addresses: []string}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "avm.listAddresses",
  "params": {
    "username": "myUsername",
    "password": "myPassword"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/x
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "addresses": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"]
  },
  "id": 1
}
```

avm.mint

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Mint units of a variable-cap asset created with [avm.createVariableCapAsset](#).

Signature:

```
avm.mint({
  amount: int,
  assetID: string,
  to: string,
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) ->
{
  txID: string,
  changeAddr: string,
}
```

- `amount` units of `assetID` will be created and controlled by address `to`.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user that pays the transaction fee. `username` must hold keys giving it permission to mint more of this asset. That is, it must control at least `threshold` keys for one of the minter sets.
- `txID` is this transaction's ID.
- `changeAddr` in the result is the address where any change was sent.

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.mint",
  "params": {
    "amount": 10000000,
    "assetID": "i1EqsthjiFTxunrj8WD2xFSrQ5p2siEKQacmCCB5qBFVqfsL2",
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
```

```

        "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
        "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
        "username": "myUsername",
        "password": "myPassword"
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2oGdPdfw2qcNUHeqjw8sU2hPVrFyNUTgn6A8HenDra7oLCDtja",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

avm.mintNFT

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Mint non-fungible tokens which were created with [avm.createNFTAsset](#).

Signature:

```

avm.mintNFT({
  assetID: string,
  payload: string,
  to: string,
  encoding: string, //optional
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) ->
{
  txID: string,
  changeAddr: string,
}

```

- `assetID` is the assetID of the newly created NFT asset.
- `payload` is an arbitrary payload of up to 1024 bytes. Its encoding format is specified by the `encoding` argument.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- `username` is the user that pays the transaction fee. `username` must hold keys giving it permission to mint more of this asset. That is, it must control at least `threshold` keys for one of the minter sets.
- `txID` is this transaction's ID.
- `changeAddr` in the result is the address where any change was sent.
- `encoding` is the encoding format to use for the payload argument. Can only be `hex` when a value is provided.

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.mintNFT",
  "params": {
    "assetID": "2KGdt2HpFKpTH5CtGzjYt5XPWs6Pv9DLoRBhiFFntbezdRvZWP",
    "payload": "0x415641204c61627338259aed",
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2oGdPdfw2qcNUHeqjw8sU2hPVrFyNUTgn6A8HenDra7oLCDtja",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

avm.send

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Send a quantity of an asset to an address.

Signature:

```
avm.send({
  amount: int,
  assetID: string,
  to: string,
  memo: string, //optional
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) -> {txID: string, changeAddr: string}
```

- Sends `amount` units of asset with ID `assetID` to address `to`. `amount` is denominated in the smallest increment of the asset. For AVAX this is 1 nAVAX (one billionth of 1 AVAX.)
- `to` is the X-Chain address the asset is sent to.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- You can attach a `memo`, whose length can be up to 256 bytes.
- The asset is sent from addresses controlled by user `username`. (Of course, that user will need to hold at least the balance of the asset being sent.)

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.send",
  "params": {
    "assetID": "AVAX",
    "amount": 10000,
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "memo": "hi, mom!",
    "username": "userThatControlsAtLeast10000OfThisAsset",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2ixSVLPNVdnFqn65rRvlrsu8WneTFqBJRMqkBjx5vZTwAQB8c1",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

avm.sendMultiple

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Sends multiple transfers of `amount` of `assetID`, to a specified address from a list of owned addresses.

Signature:

```
avm.sendMultiple({
  outputs: [],
  assetID: string,
  amount: int,
  to: string,
},
from: []string, //optional
changeAddr: string, //optional
memo: string, //optional
username: string,
password: string
}) -> {txID: string, changeAddr: string}
```

- `outputs` is an array of object literals which each contain an `assetID`, `amount` and `to`.
- `memo` is an optional message, whose length can be up to 256 bytes.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- The asset is sent from addresses controlled by user `username`. (Of course, that user will need to hold at least the balance of the asset being sent.)

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.sendMultiple",
  "params": {
    "outputs": [
      {
        "assetID": "AVAX",
        "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
        "amount": 100000000
      },
      {
        "assetID": "26aqSTpZuWDAVtRmo44fjCx4zW6PDEx3zy9Qtp2ts1MuMFn9FB",
        "to": "X-avax18knvhxx8uhc0mwlgryzjcm2wrd6e60w37xrjq",
        "amount": 10
      }
    ],
    "memo": "hi, mom!",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "username",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2iXSVLPNVdnFqn65rRvLrsu8WneTFqBJRMqkBjx5vZTwAQB8c1",
    "changeAddr": "X-avaxlturszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  }
}
```

avm.sendNFT

:::caution

Deprecated as of [v1.9.12](#).

:::

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Send a non-fungible token.

Signature:

```
avm.sendNFT({
  assetID: string,
  groupID: number,
  to: string,
  from: []string, //optional
  changeAddr: string, //optional
  username: string,
  password: string
}) -> {txID: string}
```

- `assetID` is the asset ID of the NFT being sent.
- `groupID` is the NFT group from which to send the NFT. NFT creation allows multiple groups under each NFT ID. You can issue multiple NFTs to each group.
- `to` is the X-Chain address the NFT is sent to.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed. `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- The asset is sent from addresses controlled by user `username`. (Of course, that user will need to hold at least the balance of the NFT being sent.)

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "avm.sendNFT",
  "params": {
    "assetID": "2KGdt2HpFKpTH5CtGZjYt5XPWs6Pv9DLoRBhiFfntbezdRvZWP",
    "groupID": 0,
    "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "myUsername",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X
```

Example Response:

```
{
  "jsonrpc": "2.0",
  "result": {
    "txID": "DoR2UtG1Trd3Q8gWXVevNxD666Q3DPqSFmBSMPQ9dWTV8Qtuy",
    "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
  },
  "id": 1
}
```

wallet.issueTx

Send a signed transaction to the network and assume the TX will be accepted. `encoding` specifies the format of the signed transaction. Can only be `hex` when a value is provided.

This call is made to the wallet API endpoint:

```
/ext/bc/X/wallet
```

:::caution

Endpoint deprecated as of [v1.9.12](#).

:::

Signature:

```
wallet.issueTx({
  tx: string,
  encoding: string, //optional
}) -> {
  txID: string
}
```

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "wallet.issueTx",
  "params": {

```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "result": {  
    "txID": "NUPLwbt2hsYxpQg4H2o451hmTWQ4JZx2zMzM4SinwtHgAdXlJLPHXvWSXEnepecStLj"  
  }  
}
```

wallet.send

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Send a quantity of an asset to an address and assume the TX will be accepted so that future calls can use the modified UTXO set.

This call is made to the wallet API endpoint:

/ext/bc/x/wallet

wcaution

Endpoint deprecated as of v1.9.12

• • •

Signature:

```
wallet.send({
    amount: int,
    assetID: string,
    to: string,
    memo: string, //optional
    from: [string, //optional
    changeAddr: string, //optional
    username: string,
    password: string
}) -> {txID: string, changeAddr: string}
```

- Sends `amount` units of asset with ID `assetID` to address `to`. `amount` is denominated in the smallest increment of the asset. For AVAX this is 1 nAVAX (one billionth of 1 AVAX.)
 - `to` is the X-Chain address the asset is sent to.
 - `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
 - `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
 - You can attach a `memo`, whose length can be up to 256 bytes.
 - The asset is sent from addresses controlled by user `username`. (Of course, that user will need to hold at least the balance of the asset being sent.)

Example Call:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" : "wallet.send",
  "params" : {
    "assetID" : "AVAX",
    "amount" : 10000,
    "to" : "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
    "memo" : "hi, mom!",
    "from" : "[X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5]",
    "changeAddr": "X-avax1lturszjwn05lf1pewurw96rfrd3h6x8flgs5uf8",
    "username" : "userThatControlsAtLeast100000ofThisAsset",
    "password" : "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X/wallet
```

Example Response:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "result": {  
    "txID": "2iXSVLPNVdpFnq65rBvLrsu8MaeTFcBTRMqkBJy5w7TwAOh8c1"  
  }  
}
```

```

        "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
    }
}

```

wallet.sendMultiple

:::warning Not recommended for use on Mainnet. See warning notice in [Keystore API](#). :::

Send multiple transfers of `amount` of `assetID`, to a specified address from a list of owned addresses and assume the TX will be accepted so that future calls can use the modified UTXO set.

This call is made to the wallet API endpoint:

```
/ext/bc/X/wallet
```

:::caution

Endpoint deprecated as of [v1.9.12](#).

:::

Signature:

```

wallet.sendMultiple({
  outputs: [],
  assetID: string,
  amount: int,
  to: string
},
from: []string, //optional
changeAddr: string, //optional
memo: string, //optional
username: string,
password: string
}) -> {txID: string, changeAddr: string}

```

- `outputs` is an array of object literals which each contain an `assetID`, `amount` and `to`.
- `from` are the addresses that you want to use for this operation. If omitted, uses any of your addresses as needed.
- `changeAddr` is the address any change will be sent to. If omitted, change is sent to one of the addresses controlled by the user.
- You can attach a `memo`, whose length can be up to 256 bytes.
- The asset is sent from addresses controlled by user `username`. (Of course, that user will need to hold at least the balance of the asset being sent.)

Example Call:

```

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "wallet.sendMultiple",
  "params": {
    "outputs": [
      {
        "assetID": "AVAX",
        "to": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
        "amount": 1000000000
      },
      {
        "assetID": "26aqSTpZuWDAVtRmo44fjCx4zW6PDEEx3zy9Qtp2ts1MuMFn9FB",
        "to": "X-avax18knhxx8uhc0mwlgryfyzjcm2wrd6e60w37xrjq",
        "amount": 10
      }
    ],
    "memo": "hi, mom!",
    "from": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5"],
    "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8",
    "username": "username",
    "password": "myPassword"
  }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/X/wallet

```

Example Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "txID": "2ixSVLPNVdnFqn65rRvlrsu8WneTFqBjRMqkBjx5vZTwAQb8c1",
    "changeAddr": "X-avax1turszjwn05lflpewurw96rfrd3h6x8flgs5uf8"
}
```

```
}
```

Events

Listen for transactions on a specified address.

This call is made to the events API endpoint:

```
/ext/bc/X/events
```

:::caution

Endpoint deprecated as of [v1.9.12](#).

:::

Golang Example

```
package main

import (
    "encoding/json"
    "log"
    "net"
    "net/http"
    "sync"

    "github.com/ava-labs/avalanchego/api"
    "github.com/ava-labs/avalanchego/pubsub"
    "github.com/gorilla/websocket"
)

func main() {
    dialer := websocket.Dialer{
        NetDial: func(netw, addr string) (net.Conn, error) {
            return net.Dial(netw, addr)
        },
    }

    httpHeader := http.Header{}
    conn, _, err := dialer.Dial("ws://localhost:9650/ext/bc/X/events", httpHeader)
    if err != nil {
        panic(err)
    }

    waitGroup := &sync.WaitGroup{}
    waitGroup.Add(1)

    readMsg := func() {
        defer waitGroup.Done()

        for {
            mt, msg, err := conn.ReadMessage()
            if err != nil {
                log.Println(err)
                return
            }
            switch mt {
            case websocket.TextMessage:
                log.Println(string(msg))
            default:
                log.Println(mt, string(msg))
            }
        }
    }

    go readMsg()

    cmd := &pubsub.Command{NewSet: &pubsub.NewSet{}}
    cmdmsg, err := json.Marshal(cmd)
    if err != nil {
        panic(err)
    }
    err = conn.WriteMessage(websocket.TextMessage, cmdmsg)
    if err != nil {
        panic(err)
    }
}
```

```

var addresses []string
addresses = append(addresses, " X-fuji....")
cmd = &pubsub.Command{AddAddresses: &pubsub.AddAddresses{JSONAddresses: api.JSONAddresses{Addresses: addresses}}}
cmdmsg, err = json.Marshal(cmd)
if err != nil {
    panic(err)
}

err = conn.WriteMessage(websocket.TextMessage, cmdmsg)
if err != nil {
    panic(err)
}

waitGroup.Wait()
}

```

Operations:

Command	Description	Example	Arguments
NewSet	create a new address map set	{"newSet":{}}	
NewBloom	create a new bloom set.	{"newBloom": {"maxElements":"1000","collisionProb":"0.0100"}}	maxElements - number of elements in filter must be > 0 collisionProb - allowed collision probability must be > 0 and <= 1
AddAddresses	add an address to the set	{"addAddresses":{"addresses":\[{"X-fuji..."}\]}}	addresses - list of addresses to match

Calling **NewSet** or **NewBloom** resets the filter, and must be followed with **AddAddresses**. **AddAddresses** can be called multiple times.

Set details:

- **NewSet** performs absolute address matches, if the address is in the set you will be sent the transaction.
- **NewBloom** [Bloom filtering](#) can produce false positives, but can allow a greater number of addresses to be filtered. If the addresses is in the filter, you will be sent the transaction.

Example Response:

```
2021/05/11 15:59:35 {"txID":"22HWKHrREyXyAiDnVmGp3tQQ79tHSSVxA9h26VfDEzoxvwveyk"}
```

CB58 Deprecation in AvalancheGo API

Overview

With [AvalancheGo v1.7.14 release](#), we have published changes to deprecate `cb58` encoding in favor of `hex` in the return of AvalancheGo API calls. This only impacted the encoding format for data with variable length representations (such as UTXOs, transactions, blocks, etc). Other data represented using `cb58` such as addresses and IDs (TXIDs, ChainIDs, subnetIDs, and UtxoIDs) are unchanged.

Our [AvalancheGo API documents](#) and [public API servers](#) have been updated to reflect this change: `hex` is now the default value for the `encoding` parameter in places where `cb58` used to be default.

You will need to change your code to handle the response correctly. For example, for API call [avm.getUTXOs](#) in which

- `encoding` sets the format for the returned UTXOs.

you can specify `"encoding": "hex"` when issuing the API call or leave it empty which will take the default value of `hex`.

```
curl -X POST --data '{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "avm.getUTXOs",
    "params": {
        "addresses": ["X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5", "X-avax1d09qn852zcy03sfcc9hay21lmn9hsgnw4tp3dv6"],
        "limit": 5,
        "encoding": "hex"
    }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/x
```

This gives response where UTXOs will be in `hex` format:

```
{
    "jsonrpc": "2.0",
    "result": {
        "numFetched": "5",
        "utxos": [

```

```

"0x0000a195046108a85e60f7a864bb567745a37f50c6af282103e47cc62f036cee404700000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978e
"0x0000ae8b1b9444eed8de9a81b1222f00f1b4133330add23d8ac288bffa98b85271100000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978e
"0x0000731ce04b1feefa9f4291d869adc30a33463f315491e164d89be7d6d2d7890fc00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978e
"0x0000b462030cc4734f24c0bc224cf0d16ee452ea6b67615517caffead123ab4fbf1500000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978e
"0x00005f6826c39bc957c0c6d44b70f961a99489899179cc32d21eb09c1908d7167b00000000345aa98e8a990f4101e2268fab4c4e1f731c8dfbcffa3a77978e
],
"endIndex": {
  "address": "X-avax18jma8ppw3nhx5r4ap8clazz0dps7rv5ukulre5",
  "utxo": "kbUThAUfmBXUmRgTpqD6r3nLj7rJUGho6xyht5nouNNypH45j"
},
"encoding": "hex"
},
"id": 1
}

```

You can also use `json` if the API supports it.

Affected APIs

Following APIs are affected with this change.

:::tip

When going through this API list, please make sure to check the omitted/default encoding parameter. Prior to AvalancheGo v1.7.14, by default, `cb58` is used for the `encoding` parameter if not specified in these APIs. With AvalancheGo v1.7.14 and forward, `hex` is default.

:::

X-Chain API

- [avm.buildGenesis](#)
- [avm.mintNFT](#)
- [avm.getTx](#)
- [avm.getUTXOs](#)
- [avm.issueTx](#)
- [wallet.issueTx](#)

P-Chain API

- [platform.createBlockchain](#)
- [platform.getBlock](#)
- [platform.getRewardUTXOs](#)
- [platform.getTx](#)
- [platform.getUTXOs](#)
- [platform.issueTx](#)

C-Chain API

- [avax.getAtomicTx](#)
- [avax.getUTXOs](#)
- [avax.issueTx](#)

Index API

- [index.getLastAccepted](#)
- [index.getContainerByIndex](#)
- [index.getContainerByID](#)
- [index.getContainerRange](#)
- [index.getIndex](#)
- [index.isAccepted](#)

Keystore API

- [keystore.exportUser](#)
- [keystore.importUser](#)

description: Postman is a free tool used by developers to quickly and easily send REST, SOAP, and GraphQL requests and test APIs.

Postman Collection

What Is Postman?

Postman is a free tool used by developers to quickly and easily send REST, SOAP, and GraphQL requests and test APIs. It is available as both an online tool and an application for Linux, MacOS and Windows. Postman allows you to quickly issue API calls and see the responses in a nicely formatted, searchable form.

We have made a Postman collection for Avalanche, that includes all the public API calls that are available on [AvalancheGo Instance](#), allowing you to quickly issue commands to your node and see the response, without having to copy and paste long and complicated `curl` commands.

Along with the API collection, there is also the example Avalanche environment for Postman, that defines common variables such as IP address of the node, your Avalanche addresses and similar common elements of the queries, so you don't have to enter them multiple times.

Combined, they will allow you to easily keep tabs on your node, check on its state and do quick queries to find out details about its operation.

Setup

Postman Installation

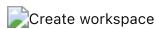
Postman can be installed locally or used as a web app. We recommend installing the application, as it simplifies operation. You can download Postman from its [website](#). It is recommended that you sign up using your email address as then your workspace can be easily backed up and shared between web app and the app installed on your computer.



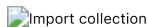
After you installed the application, run it. It will prompt you to create an account or log in. Do so. Again, it is not necessary, but recommended.

Collection Import

Select `Create workspace` from Workspaces tab and follow the prompts to create a new workspace. This will where the rest of the work will be done.



We're ready to import the collection. On the top-left corner of the Workspaces tab select `Import` and switch to `Link` tab.



There, in the URL input field paste the link to the collection:

```
https://raw.githubusercontent.com/ava-labs/avalanche-postman-collection/master/Avalanche.postman_collection.json
```

Postman will recognize the format of the file content and offer to import the file as a collection. Complete the import. Now you will have Avalanche collection in your Workspace.



Environment Import

Next, we have to import the environment variables. Again, on the top-left corner of the Workspaces tab select `Import` and switch to `Link` tab. This time, paste the link to the environment JSON:

```
https://raw.githubusercontent.com/ava-labs/avalanche-postman-collection/master/Example-Avalanche-Environment.postman_environment.json
```

Postman will recognize the format of the file:



Import it to your workspace. Now, we will need to edit that environment to suit the actual parameters of your particular installation. These are the parameters that differ from the defaults in the imported file.

Select the Environments tab, choose the Avalanche environment which was just added. You can directly edit any values here:



As a minimum, you will need to change the IP address of your node, which is the value of the `host` variable. Change it to the IP of your node (change both the `initial` and `current` values). Also, if your node is not running on the same machine where you installed Postman, make sure your node is accepting the connections on the API port from the outside by checking the appropriate [command line option](#).

Now we sorted everything out, and we're ready to query the node.

Making API Calls

Open one of the API call groups, for example `Health`. Click `health` call:



You will see that format of the call uses the `baseURL` environment variable. Click `Send`. Request will be sent, and soon you will see the response, in the `Body` tab in the `Response`:



To see the actual call and the variables that are sent to the node, switch to `Body` tab in the API call tabs. There you can quickly change the variables to see the response to different queries.

Conclusion

If you completed the tutorial, you are now able to quickly issue API calls to your node without messing with the curl commands in the terminal. This allows you to quickly see the state of your node, track changes or double-check the health or liveness of your node.

Contributing

We're hoping to continuously keep this collection up-to-date with the [Avalanche APIs](#), and also add [data visualizations](#). If you're able to help improve the Avalanche Postman Collection in any way, first create a feature branch by branching off of `master`, next make the improvements on your feature branch and lastly create a [pull request](#) to merge your work back in to `master`.

If you have any other questions or suggestions, come [talk to us](#).

description: There is a public API server that allows developers to access the Avalanche platform without having to run a node themselves.

Public API Server

There is a public API server that allows developers to access the Avalanche network without having to run a node themselves. The public API server is actually several [AvalancheGo](#) nodes behind a load balancer to ensure high availability and high request throughput.

Using the Public API Nodes

The public API server is at `api.avax.network` for Avalanche Mainnet and `api.avax-test.network` for Avalanche Fuji Testnet. To access a particular API, just append the relevant API endpoint, as documented [here](#). Namely, use the following end points for each chain respectively:

HTTP

- For C-Chain API, the URL is `https://api.avax.network/ext/bc/C/rpc`.
- For X-Chain API, the URL is `https://api.avax.network/ext/bc/X`.
- For P-Chain API, the URL is `https://api.avax.network/ext/bc/P`.

Note: on Fuji Testnet, use `https://api.avax-test.network/` instead of `https://api.avax.network/`.

WebSocket

- For C-Chain API, the URL is `wss://api.avax.network/ext/bc/C/ws`.

Note: on Fuji Testnet, the URL is `wss://api.avax-test.network/ext/bc/C/ws`.

Supported APIs

The public API server supports all the API endpoints that make sense to be available on a public-facing service, including APIs for the [X-Chain](#), [P-Chain](#), [C-Chain](#), and full archival for the Primary Network. However, it doesn't support [Index APIs](#), which includes the X-Chain API's `getAddressTxs` method.

For a full list of available APIs see [here](#).

Limitations

The public API only supports C-Chain WebSocket API calls for API methods that don't exist on the C-Chain's HTTP API.

For batched C-Chain requests on the public API node, the maximum number of items is 40. We're working on to support a larger batch size.

The maximum number of blocks to serve per `getLogs` request is 2048, which is set by `api-max-blocks-per-request`.

Sticky Sessions

Requests to the public API server API are distributed by a load balancer to an individual node. As a result, consecutive requests may go to different nodes. That can cause issues for some use cases. For example, one node may think a given transaction is accepted, while for another node the transaction is still processing. To work around this, you can use 'sticky sessions', as documented [here](#). This allows consecutive API calls to be routed to the same node.

If you're using [AvalancheJS](#) to access the public API, simply set the following in your code:

```
avalanche.setRequestConfig("withCredentials", true)
```

Availability

Usage of public API nodes is free and available to everyone without any authentication or authorization. Rate limiting is present, but many of the API calls are cached, and the rate limits are quite high. If your app is running up against the limits, please [contact us](#) or try using a community RPC provider.

Support

If you have questions, problems, or suggestions, join the official [Avalanche Discord](#).

Subnets

DeFi Kingdom (DFK)

HTTP

- The URL is `https://subnets.avax.network/defi-kingdoms/dfk-chain/rpc`.

Note: on Fuji Testnet, the URL is `https://subnets.avax.network/defi-kingdoms/dfk-chain-testnet/rpc`.

Websockets

- The URL is `wss://subnets.avax.network/defi-kingdoms/dfk-chain/ws`.

Note: on Fuji Testnet, the URL is `wss://subnets.avax.network/defi-kingdoms/dfk-chain-testnet/ws`.

Dexalot

HTTP

- The URL is `https://subnets.avax.network/dexalot/mainnet/rpc`.

Note: on Fuji Testnet, the URL is `https://subnets.avax.network/dexalot/testnet/rpc`.

Websockets

- The URL is `wss://subnets.avax.network/dexalot/mainnet/ws`.

Note: on Fuji Testnet, the URL is `wss://subnets.avax.network/dexalot/testnet/ws`.

Community Providers

:::warning Disclaimer

Provided for informational purposes only, without representation, warranty or guarantee of any kind. None of this is as an endorsement by the Avalanche Foundation Limited, Ava Labs, Inc. or any of their respective subsidiaries or affiliates, nor is any of this investment or financial advice. Please review this [Notice](#) and conduct your own research to properly evaluate the risks and benefits of any project.

:::

Infura

[Infura](#) currently only supports the C-Chain.

HTTP

- For C-Chain API, the URL is `https://avalanche-mainnet.infura.io/v3/YOUR-API-KEY`

Note: on Fuji Testnet, the URL is `https://avalanche-fuji.infura.io/v3/YOUR-API-KEY`.

ANKR

Mainnet

- Standard EVM API, the URL is `https://rpc.ankr.com/avalanche`.
- For C-Chain API, the URL is `https://rpc.ankr.com/avalanche-c`. On ANKR the C-Chain API doesn't support standard EVM APIs. For that use the Standard EVM API.
- For X-Chain API, the URL is `https://rpc.ankr.com/avalanche-x`.
- For P-Chain API, the URL is `https://rpc.ankr.com/avalanche-p`.

Testnet (Fuji)

- Standard EVM API, the URL is `https://rpc.ankr.com/avalanche_fuji`.
- For C-Chain API, the URL is `https://rpc.ankr.com/avalanche_fuji-c`. On ANKR the C-Chain API doesn't support standard EVM APIs. For that use the Standard EVM API.
- For X-Chain API, the URL is `https://rpc.ankr.com/avalanche_fuji-x`.
- For P-Chain API, the URL is `https://rpc.ankr.com/avalanche_fuji-p`.

Features:

- Archive Data Included.
- Automatic geo-routing across North America, Europe, and Asia.

Note: soft limited to 1 million daily requests per IP or referring domain. Batch calls limited to 1000.

Support is available on the [ANKR Discord](#).

Blast

[Blast](#) supports the C-Chain, X-Chain, and P-Chain.

Mainnet

HTTP

- For C-Chain RPC Endpoint ETH, the URL is `https://ava-mainnet.public.blastapi.io/ext/bc/C/rpc`

- For C-Chain RPC Endpoint AVAX, the URL is `https://ava-mainnet.public.blastapi.io/ext/bc/C/avax`
- For X-Chain RPC Endpoint, the URL is `https://ava-mainnet.public.blastapi.io/ext/bc/X`
- For P-Chain RPC Endpoint, the URL is `https://ava-mainnet.public.blastapi.io/ext/P`

Websockets

- For C-Chain WSS Endpoint, the URL is `wss://ava-mainnet.public.blastapi.io/ext/bc/C/ws`

Testnet (Fuji)

HTTP

- For C-Chain RPC Endpoint ETH, the URL is `https://ava-testnet.public.blastapi.io/ext/bc/C/rpc`
- For C-Chain RPC Endpoint AVAX, the URL is `https://ava-testnet.public.blastapi.io/ext/bc/C/avax`
- For X-Chain RPC Endpoint, the URL is `https://ava-testnet.public.blastapi.io/ext/bc/X`
- For P-Chain RPC Endpoint, the URL is `https://ava-testnet.public.blastapi.io/ext/P`

Websockets

- For C-Chain WSS Endpoint, the URL is `wss://ava-testnet.public.blastapi.io/ext/bc/C/ws`

BlockSpaces

[BlockSpaces](#) supports the C-Chain.

HTTP

- For C-Chain RPC Endpoint AVAX, the URL is `https://web3endpoints.com/avax-mainnet`

GetBlock

[GetBlock](#) currently only supports the C-Chain.

HTTP

- For C-Chain API, the URL is `https://avax.getblock.io/api_key/mainnet/ext/bc/C/ws?api_key=`

Note: on Fuji Testnet, the URL is `https://avax.getblock.io/api_key/testnet/ext/bc/C/ws?api_key=`.

Websockets

- For C-Chain API, the URL is `wss://avax.getblock.io/api_key/mainnet/ext/bc/C/ws?api_key=`

Note: on Fuji Testnet, the URL is `wss://avax.getblock.io/api_key/testnet/ext/bc/C/ws?api_key=`.

QuickNode

[QuickNode](#) supports the X-Chain, P-Chain, C-Chain, and Index API.

HTTP

- The URL is `http://sample-endpoint-name.network.quiknode.pro/token-goes-here/`

Websockets

- The URL is `wss://sample-endpoint-name.network.quiknode.pro/token-goes-here/`

ChainStack

[ChainStack](#) supports the C-Chain, X-Chain, P-Chain, and the Fuji Testnet.

Features:

- Globally distributed infrastructure for optimal performance.
- Crypto payments natively.
- 24/7 customer support.

Mainnet

HTTP

- For C-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/C/rpc`
- For X-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/X`
- For P-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/P`

Websockets

Websockets are available for the C-chain and the X-chain.

- For C-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/C/ws`
- For X-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/X/events`

Testnet (Fuji)

HTTP

- For C-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/C/rpc`
- For X-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/X`
- For P-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/P`

Websockets

Websockets are available for the C-chain and the X-chain.

- For C-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/C/ws`
- For X-Chain API, the URL is `https://nd-123-145-789.p2pify.com/API_KEY/ext/bc/X/events`

PublicNode by Allnodes

[PublicNode](#) supports the C-Chain on the Mainnet.

Features:

- Free
- Privacy oriented
- Globally distributed infrastructure on Allnodes
- Optimized for speed and reliability - check our page for stats

Mainnet

HTTPS

- For C-Chain API, the URL is <https://avalanche-c-chain.publicnode.com>

X-Chain Migration Guide

Overview

This document summarizes all of the changes made to the X-Chain API to support Avalanche Cortina (v1.10.0), which migrates the X-Chain to run Snowman++. In summary, the core transaction submission and confirmation flow is unchanged, however, there are new APIs that must be called to index all transactions.

Transaction Broadcast and Confirmation

The transaction format on the X-Chain does not change in Cortina. This means that wallets that have already integrated with the X-Chain don't need to change how they sign transactions. Additionally, there is no change to the format of the [avm.issueTx](#) or the [avm.getTx](#) API.

However, the [avm.getStatus](#) endpoint is now deprecated and its usage should be replaced with [avm.getTx](#) (which only returns accepted transactions for AvalancheGo >= v1.9.12). [avm.getStatus](#) will still work up to and after the Cortina activation if you wish to migrate after the network upgrade has occurred.

Vertex -> Block Indexing

Before Cortina, indexing the X-Chain required polling the [/ext/index/X/vtx](#) endpoint to fetch new vertices. During the Cortina activation, a "stop vertex" will be produced using a [new codec version](#) that will contain no transactions. This new vertex type will be the [same format](#) as previous vertices. To ensure historical data can still be accessed in Cortina, the [/ext/index/X/vtx](#) will remain accessible even though it will no longer be populated with chain data.

:::note

The index for the X-chain tx and vtx endpoints will never increase again. The index for the X-chain blocks will increase as new blocks are added.

:::

After Cortina activation, you will need to migrate to using the new [ext/index/X/block](#) endpoint (shares the same semantics as [/ext/index/P/block](#)) to continue indexing X-Chain activity. Because X-Chain ordering is deterministic in Cortina, this means that X-Chain blocks across all heights will be consistent across all nodes and will include a timestamp. Here is an example of iterating over these blocks in Golang:

```
package main

import (
    "context"
    "fmt"
    "log"
    "time"

    "github.com/ava-labs/avalanchego/indexer"
    "github.com/ava-labs/avalanchego/vms/proposervm/block"
    "github.com/ava-labs/avalanchego/wallet/chain/x"
    "github.com/ava-labs/avalanchego/wallet/subnet/primary"
)

func main() {
    var (
        uri      = fmt.Sprintf("%s/ext/index/X/block", primary.LocalAPIURI)
        client   = indexer.NewClient(uri)
        ctx      = context.Background()
        nextIndex uint64
    )
    for {
        log.Printf("polling for next accepted block")
        container, err := client.GetContainerByIndex(ctx, nextIndex)
        if err != nil {
            time.Sleep(time.Second)
            continue
        }

        proposerVMBlock, err := block.Parse(container.Bytes)
        if err != nil {
            log.Fatalf("failed to parse proposervm block: %s\n", err)
        }
    }
}
```

```

        avmBlockBytes := proposerVMBlock.Block()
        avmBlock, err := x.Parser.ParseBlock(avmBlockBytes)
        if err != nil {
            log.Fatalf("failed to parse avm block: %s\n", err)
        }

        acceptedTxs := avmBlock.Txs()
        log.Printf("accepted block %s with %d transactions", avmBlock.ID(), len(acceptedTxs))

        for _, tx := range acceptedTxs {
            log.Printf("accepted transaction %s", tx.ID())
        }

        nextIndex++
    }
}

```

After Cortina activation, it will also be possible to fetch X-Chain blocks directly without enabling the Index API. You can use the [avm.getBlock](#), [avm.getBlockByHeight](#), and [avm.getHeight](#) endpoints to do so. This, again, will be similar to the [P-Chain semantics](#).

Deprecated API Calls

This long-term deprecation effort will better align usage of AvalancheGo with its purpose, to be a minimal and efficient runtime that supports only what is required to validate the Primary Network and Subnets. Integrators should make plans to migrate to tools and services that are better optimized for serving queries over Avalanche Network state and avoid keeping any keys on the node itself.

:::note

This deprecation ONLY applies to APIs that AvalancheGo exposes over the HTTP port. Transaction types with similar names to these APIs are NOT being deprecated.

:::

- ipcs
 - ipcs.publishBlockchain
 - ipcs.unpublishBlockchain
 - ipcs.getPublishedBlockchains
- keystore
 - keystore.createUser
 - keystore.deleteUser
 - keystore.listUsers
 - keystore.importUser
 - keystore.exportUser
- avm/pubsub
- avm
 - avm.getAddressTxs
 - avm.getBalance
 - avm.getAllBalances
 - avm.createAsset
 - avm.createFixedCapAsset
 - avm.createVariableCapAsset
 - avm.createNFTAsset
 - avm.createAddress
 - avm.listAddresses
 - avm.exportKey
 - avm.importKey
 - avm.mint
 - avm.sendNFT
 - avm.mintNFT
 - avm.import
 - avm.export
 - avm.send
 - avm.sendMultiple
- avm/wallet
 - wallet.issueTx
 - wallet.send
 - wallet.sendMultiple
- platform
 - platform.exportKey
 - platform.importKey
 - platform.getBalance
 - platform.createAddress
 - platform.listAddresses
 - platform.getSubnets
 - platform.addValidator

- platform.addDelegator
- platform.addSubnetValidator
- platform.createSubnet
- platform.exportAVAX
- platform.importAVAX
- platform.createBlockchain
- platform.getBlockchains
- platform.getStake
- platform.getMaxStakeAmount
- platform.getRewardUTXOs

Cortina FAQ

Do I Have to Upgrade my Node?

If you don't upgrade your validator to `v1.10.0` before the Avalanche Mainnet activation date, your node will be marked as offline and other nodes will report your node as having lower uptime, which may jeopardize your staking rewards.

Is There any Change in Hardware Requirements?

No.

Will Updating Decrease my Validator's Uptime?

No. As a reminder, you can check your validator's estimated uptime using the [info.uptime API call](#).

I Think Something Is Wrong. What Should I Do?

First, make sure that you've read the documentation thoroughly and checked the [FAQs](#). If you don't see an answer to your question, go to our [Discord](#) server and search for your question. If it has not already been asked, please post it in the appropriate channel.

AvalancheJS Overview

AvalancheJS is a JavaScript Library for interfacing with the [Avalanche](#) platform. It is built using TypeScript and intended to support both browser and Node.js. The AvalancheJS library allows one to issue commands to the Avalanche node APIs.

The APIs currently supported by default are:

- Admin API
- Auth API
- AVM API (X-Chain)
- EVM API (C-Chain)
- Health API
- Index API
- Info API
- Keystore API
- Metrics API
- PlatformVM API
- Socket API

We built AvalancheJS with ease of use in mind. With this library, any JavaScript developer is able to interact with a node on the Avalanche Platform who has enabled their API endpoints for the developer's consumption. We keep the library up-to-date with the latest changes in the Avalanche Platform Specification found in the [Platform Chain Specification](#), [Exchange Chain \(X-Chain\) Specification](#), [Contract Chain \(C-Chain\) Specification](#).

Using AvalancheJS, developers can:

- Locally manage private keys
- Retrieve balances on addresses
- Get UTXOs for addresses
- Build and sign transactions
- Issue signed transactions to the X-Chain, P-Chain and C-Chain on the Primary network
- Create a Subnetwork
- Swap AVAX and assets between the X-Chain, P-Chain and C-Chain
- Add a Validator to the Primary network
- Add a Delegator to the Primary network
- Administer a local node
- Retrieve Avalanche network information from a node

Requirements

AvalancheJS requires Node.js version 14.18.0 or higher to compile.

Installation

Avalanche is available for install via `npm` or `yarn`. For installing via `npm`:

```
npm install --save avalanche
```

For installing via `yarn`:

```
yarn add avalanche
```

You can also pull the repo down directly and build it from scratch:

```
npm run build
```

This will generate a pure JavaScript library and place it in a folder named "web" in the project root. The "avalanchejs" file can then be dropped into any project as a pure JavaScript implementation of Avalanche.

The AvalancheJS library can be imported into your existing Node.js project as follows:

```
const avalanche = require("avalanche")
```

Or into your TypeScript project like this:

```
import { Avalanche } from "avalanche"
```

Importing Essentials

```
import { Avalanche, BinTools, Buffer, BN } from "avalanche"

let bintoools = BinTools.getInstance()
```

The above lines import the libraries used in the tutorials. The libraries include:

- `avalanche`: Our JavaScript module.
- `bn.js`: A big number module use by AvalancheJS.
- `buffer`: A Buffer library.
- `BinTools`: A singleton built into AvalancheJS that is used for dealing with binary data.

API

AvalancheJS - [V3.15.0](#)

Modules

- [AVM](#)
 - [AVM BaseTx](#)
 - [AVM Constants](#)
 - [AVM CreateAssetTx](#)
 - [AVM Credentials](#)
 - [AVM ExportTx](#)
 - [AVM GenesisAsset](#)
 - [AVM GenesisData](#)
 - [AVM ImportTx](#)
 - [AVM InitialStates](#)
 - [AVM Inputs](#)
 - [AVM KeyChain](#)
 - [AVM MinterSet](#)
 - [AVM OperationTx](#)
 - [AVM Operations](#)
 - [AVM Outputs](#)
 - [AVM Transactions](#)
 - [AVM UTXOs](#)
 - [AVM Vertex](#)
- [Admin](#)
- [Auth](#)
- [EVM](#)
 - [EVM BaseTx](#)
 - [EVM Constants](#)
 - [EVM Credentials](#)
 - [EVM ExportTx](#)
 - [EVM ImportTx](#)
 - [EVM Inputs](#)
 - [EVM KeyChain](#)
 - [EVM Outputs](#)
 - [EVM Transactions](#)
 - [EVM UTXOs](#)
- [Health](#)
- [Index](#)
- [Info](#)
- [Keystore](#)
- [Metrics](#)
- [PlatformVM](#)

- [PlatformVM AddSubnetValidatorTx](#)
- [PlatformVM BaseTx](#)
- [PlatformVM Constants](#)
- [PlatformVM CreateChainTx](#)
- [PlatformVM CreateSubnetTx](#)
- [PlatformVM Credentials](#)
- [PlatformVM ExportTx](#)
- [PlatformVM ImportTx](#)
- [PlatformVM Inputs](#)
- [PlatformVM KeyChain](#)
- [PlatformVM Outputs](#)
- [PlatformVM SubnetAuth](#)
- [PlatformVM Transactions](#)
- [PlatformVM UTXOs](#)
- [PlatformVM ValidationTx](#)
- [Socket](#)
- [AVM Interfaces](#)
- [Admin Interfaces](#)
- [Auth Interfaces](#)
- [Avalanche](#)
- [AvalancheCore](#)
- [Common API Base](#)
- [Common AssetAmount](#)
- [Common Inputs](#)
- [Common Interfaces](#)
- [Common JRPCAPI](#)
- [Common KeyChain](#)
- [Common NBytes](#)
- [Common Output](#)
- [Common RESTAPI](#)
- [Common SECP256k1KeyChain](#)
- [Common Signature](#)
- [Common Transactions](#)
- [Common UTXOs](#)
- [EVM Interfaces](#)
- [Health Interfaces](#)
- [Index Interfaces](#)
- [Info Interfaces](#)
- [Keystore Interfaces](#)
- [PlatformVM Interfaces](#)
- [Utils Base68](#)
- [Utils BinTools](#)
- [Utils Constants](#)
- [Utils DB](#)
- [Utils HDNode](#)
- [Utils HelperFunctions](#)
- [Utils Mnemonic](#)
- [Utils Payload](#)
- [Utils PersistanceOptions](#)
- [Utils Serialization](#)
- [src/apis/admin](#)
- [src/apis/auth](#)
- [src/apis/avm](#)
- [src/apis/evm](#)
- [src/apis/health](#)
- [src/apis/index](#)
- [src/apis/info](#)
- [src/apis/keystore](#)
- [src/apis/metrics](#)
- [src/apis/platformvm](#)
- [src/apis/socket](#)
- [src/common](#)
- [src/utils](#)

Create an Asset on the X-Chain

This example creates an asset on the X-Chain and publishes it to the Avalanche platform. The first step in this process is to create an instance of AvalancheJS connected to our Avalanche platform endpoint of choice. In this example we're using the local network 12345 via [Avalanche Network Runner](#). The code examples are written in typescript. The script is in full, in both typescript and JavaScript, after the individual steps. The whole example can be found [here](#).

```
import { Avalanche, BN, Buffer } from "avalanche"
import {
  AVMAPI,
  KeyChain,
  UTXOSET,
```

```

UnsignedTx,
Tx,
InitialStates,
SECPMintOutput,
SECPTransferOutput,
} from "avalanche/dist/apis/avm"
import {
  PrivateKeyPrefix,
  DefaultLocalGenesisPrivateKey,
} from "avalanche/dist/utils"

const ip: string = "localhost"
const port: number = 9650
const protocol: string = "http"
const networkID: number = 12345 // Default is 1, we want to override that for our local network
const avalanche: Avalanche = new Avalanche(ip, port, protocol, networkID)

```

Import the Local Network's Pre-funded Address

Next we get an instance of the X-Chain local keychain. The local network 12345 has a pre-funded address which you can access with the private key

```

PrivateKeyKey-ewoqjP7PxY4yr3iLTpLisriqt94hdyDFNgchSxGGztUrTXtNN
${PrivateKeyPrefix}${DefaultLocalGenesisPrivateKey} . Lastly get the pre-funded address as a Buffer and as a string .

```

```

const xchain: AVMAPI = avalanche.XChain()
const xKeychain: KeyChain = xchain.keyChain()
const privKey: string = `${PrivateKeyPrefix}${DefaultLocalGenesisPrivateKey}`
xKeychain.importKey(privKey)
const xAddresses: Buffer[] = xchain.keyChain().getAddresses()
const xAddressStrings: string[] = xchain.keyChain().getAddressStrings()

```

Prepare for the Mint Output

Now we need to create an empty array for the SECPMintOutput . We also need a threshold and locktime for the outputs which we're going to create. Each X-Chain transaction can contain a memo field of up to 256 bytes of arbitrary data.

```

const outputs: SECPMintOutput[] = []
const threshold: number = 1
const locktime: BN = new BN(0)
const memo: Buffer = Buffer.from(
  "AVM utility method buildCreateAssetTx to create an ANT"
)

```

Describe the New Asset

The first step in creating a new asset using AvalancheJS is to determine the qualities of the asset. We will give the asset a name, a ticker symbol, as well as a denomination.

```

const name: string = "TestToken"
const symbol: string = "TEST"
const denomination: number = 3

```

Set Up Async / Await

The remaining code will be encapsulated by this main function so that we can use the async / await pattern.

```

const main = async (): Promise<any> => {
  main()
}

```

Fetch the UTXO

Pass the xAddressStrings to xchain.getUTXOs to fetch the UTXO.

```

const avmUTXOResponse: any = await xchain.getUTXOs(xAddressStrings)
const utxoSet: UTXOSet = avmUTXOResponse.utxos

```

Creating the Initial State

We want to mint an asset with 507 units held by the managed key. This sets up the state that will result from the Create Asset transaction.

```

// Create outputs for the asset's initial state
const amount: BN = new BN(507)

```

```

const vcapSepcOutput: SECPTransferOutput = new SECPTransferOutput(
  amount,
  xAddresses,
  locktime,
  threshold
)
const initialStates: InitialStates = new InitialStates()

// Populate the initialStates with the outputs
initialStates.addOutput(vcapSepcOutput)

```

Create the Mint Output

We also want to create a `SECPMintOutput` so that we can mint more of this asset later.

```

const secpMintOutput: SECPMintOutput = new SECPMintOutput(
  xAddresses,
  locktime,
  threshold
)
outputs.push(secpMintOutput)

```

Creating the Signed Transaction

Now that we know what we want an asset to look like, we create a transaction to send to the network. There is an AVM helper function `buildCreateAssetTx()` which does just that.

```

const unsignedTx: UnsignedTx = await xchain.buildCreateAssetTx(
  utxoSet,
  xAddressStrings,
  xAddressStrings,
  initialStates,
  name,
  symbol,
  denomination,
  outputs,
  memo
)

```

Sign and Issue the Transaction

Now let's sign the transaction and issue it to the Avalanche network. If successful it will return a [CB58](#) serialized string for the transaction ID.

Now that we have a signed transaction ready to send to the network, let's issue it!

```

const tx: Tx = unsignedTx.sign(xKeychain)
const txid: string = await xchain.issueTx(tx)
console.log(`Success! TXID: ${txid}`)

```

Get the Status of the Transaction

Now that we sent the transaction to the network, it takes a few seconds to determine if the transaction has gone through. We can get an updated status on the transaction using the transaction ID through the AVM API.

```

// returns one of: "Accepted", "Processing", "Unknown", and "Rejected"
const status: string = await xchain.getTxStatus(id)

```

The statuses can be one of `Accepted`, `Processing`, `Unknown`, and `Rejected`

- "Accepted" indicates that the transaction has been accepted as valid by the network and executed
- "Processing" indicates that the transaction is being voted on.
- "Unknown" indicates that node knows nothing about the transaction, indicating the node doesn't have it
- "Rejected" indicates the node knows about the transaction, but it conflicted with an accepted transaction

Identifying the Newly Created Asset

The X-Chain uses the transaction ID of the transaction which created the asset as the unique identifier for the asset. This unique identifier is henceforth known as the "AssetID" of the asset. When assets are traded around the X-Chain, they always reference the AssetID that they represent.

Generate a TxID Using AvalancheJS

Introduction

A transaction id, or TxID, is a string of numbers and letters which identifies a specific transaction on the blockchain. TXIDs are one of the most core components that developers interact with when working on a blockchain system. They're deterministic and can be generated using [AvalancheGo](#) or [AvalancheJS](#).

On the Avalanche network a TxID is a CB58 encoded string which is created by `sha256` hashing the transaction. `CB58` is a base58 encoding with a checksum. Below are the steps for deterministically generating a TxID using AvalancheJS.

Creating a Signed Transaction

The script which we're using can be found in the AvalancheJS example scripts as [example/avm/baseTx-AVAX.ts](#). First, create a `BaseTx`. We're omitting that part for the sake of brevity. Once you have a `BaseTx` then create an `UnsignedTx` and sign it. Lastly convert the `Tx` to a `Buffer` by calling `tx.toBuffer()`.

```
// Manually build BaseTx via steps in example/avm/baseTx-avax.ts
// Create an UnsignedTx with the BaseTx
const unsignedTx: UnsignedTx = new UnsignedTx(baseTx)
// Sign the UnsignedTx to create a Tx
const tx: Tx = unsignedTx.sign(xKeychain)
// Convert the Tx to a Buffer
const txBuf: Buffer = tx.toBuffer()
```

Generate the TxID

Next, create a `sha256` hash of the `Buffer` from the previous step.

```
// Create sha256 hash of the Tx Buffer
const sha256Hash: Buffer = Buffer.from(
  createHash("sha256").update(txBuf).digest().buffer
)
```

As mentioned in the [Introduction](#), a TxID is a CB58 encoded string which is created by `sha256` hashing the transaction. To create the TxID now CB58 encode the newly created `sha256` hash.

```
// cb58 the sha256 hash
const generatedTxID: string = bintools.cb58Encode(sha256Hash)
console.log(`Generated TxID: ${generatedTxID}`)
```

The `generatedTxID` will be a CB58 encoded string similar to `eLXEKFFMgGmK7ZLokCFjppdBfGy5hDuRqh5uJVyXXPaRErpAX`.

Confirm TxID Is Correct

To confirm that the `generatedTxID` is correct issue the `BaseTx` to AvalancheGo and compare the TxID which is returned with the recently created TxID.

```
// get the actual txID from the full node
const actualTxID: string = await xchain.issueTx(tx)
console.log(`Success! TxID: ${actualTxID}`)

// Note the generated TxID and the returned TxID match
Generated TXID: eLXEKFFMgGmK7ZLokCFjppdBfGy5hDuRqh5uJVyXXPaRErpAX
Returned TXID: eLXEKFFMgGmK7ZLokCFjppdBfGy5hDuRqh5uJVyXXPaRErpAX
```

Summary

TXIDs are a core component of any blockchain system. They are used extensively in Avalanche when creating transactions, issuing new assets and even spinning up Subnets and validators. TXIDs are deterministically created by `sha256` hashing a `Buffer` of the transaction and then CB58 encoding the hash.

Manage X-Chain Keys

AvalancheJS comes with its own AVM Keychain. This KeyChain is used in the functions of the API, enabling them to sign using keys it's registered. The first step in this process is to create an instance of AvalancheJS connected to our Avalanche platform endpoint of choice.

```
import { Avalanche, BinTools, Buffer, BN } from "avalanche"

let bintools = BinTools.getInstance()

let myNetworkID = 12345 //default is 1, we want to override that for our local network
let myBlockchainID = "GJABrZ9A6UQFpwjPU8MDxDd8vuyRoDVeDAXc694wJ5t3zEkhU" // The X-Chain blockchainID on this network
let ava = new avalanche.Avalanche(
  "localhost",
  9650,
  "http",
  myNetworkID,
  myBlockchainID
```

```
)  
let xchain = ava.XChain() //returns a reference to the X-Chain used by AvalancheJS
```

Accessing the Keychain

The KeyChain is accessed through the X-Chain and can be referenced directly or through a reference variable.

```
let myKeychain = xchain.keyChain()
```

This exposes the instance of the class AVMKeyChain which is created when the X-Chain API is created. At present, this supports secp256k1 curve for ECDSA key pairs.

Creating X-Chain Key Pairs

The KeyChain has the ability to create new Keypairs for you and return the address associated with the key pair.

```
let newAddress1 = myKeychain.makeKey() //returns a Buffer for the address
```

You may also import your existing private key into the KeyChain using either a Buffer...

```
let mypk = bintools.avaDeserialize(  
  "24juJ9vZexUM6expymcT48LBx27klm7xpaoV62oSQAHDziao5"  
) //returns a Buffer  
let newAddress2 = myKeychain.importKey(mypk) //returns a Buffer for the address
```

... or an Avalanche serialized string works, too:

```
let mypk = "24juJ9vZexUM6expymcT48LBx27klm7xpaoV62oSQAHDziao5"  
let newAddress2 = myKeychain.importKey(mypk) //returns a Buffer for the address
```

Working with Keychains

The X-Chain's KeyChain has standardized key management capabilities. The following functions are available on any KeyChain that implements this interface.

```
let addresses = myKeychain.getAddresses(); //returns an array of Buffers for the addresses  
let addressStrings = myKeychain.getAddressStrings(); //returns an array of strings for the addresses  
let exists = myKeychain.hasKey(newAddress1); //returns true if the address is managed  
let keypair = myKeychain.getKey(newAddress1); //returns the KeyPair class
```

Working with Keypairs

The X-Chain's keypair has standardized keypair functionality. The following operations are available on any keypair that implements this interface.

```
let address = keypair.getAddress() //returns Buffer  
let addressString = keypair.getAddressString() //returns string  
  
let pubk = keypair.getPublicKey() //returns Buffer  
let pubkstr = keypair.getPublicKeyString() //returns a CB58 encoded string  
  
let privk = keypair.getPrivateKey() //returns Buffer  
let privkstr = keypair.getPrivateKeyString() //returns a CB58 encoded string  
  
keypair.generateKey() //creates a new random KeyPair  
  
let mypk = "24juJ9vZexUM6expymcT48LBx27klm7xpaoV62oSQAHDziao5"  
let successul = keypair.importKey(mypk) //returns boolean if private key imported successfully  
  
let message = Buffer.from("Wubalubadubdub")  
let signature = keypair.sign(message) //returns a Buffer with the signature  
  
let signerPubk = keypair.recover(message, signature)  
let isValid = keypair.verify(message, signature) //returns a boolean
```

Encode Bech32 Addresses

The X-Chain and the P-Chain use Bech32 to encode addresses. Note, the C-Chain also uses Bech32 to encode addresses for importing and exporting assets however the EVM, in general, uses hex encoding for addresses. Ex: `0x46f3e64E4e3f5a46Eaf5c292301c6174B9B646BF`.

Each Bech32 address is composed of the following components

1. A Human-Readable Part (HRP).
2. The number `1` is a separator (the last digit `1` seen is considered the separator).
3. Base-32 encoded string for the data part of the address (the 20-byte address itself).

4. A 6-character base-32 encoded error correction code using the BCH algorithm.

For example the following Bech32 address, `X-avax19rknw8l0grnfunjrzwxlxync6zrlu33y2jxhrg`, is composed like so:

1. HRP: `avax`
2. Separator: `1`
3. Address: `9rknw8l0grnfunjrzwxlxync6zrlu33y`
4. Checksum: `2jxhrg`

Depending on the `networkID` which is passed in when instantiating `Avalanche` the encoded addresses will have a distinctive HRP per each network.

`AvalancheJS` also has address encoding for past networks `cascade`, `denali`, and `everest`.

- 0 - X- custom `19rknw8l0grnfunjrzwxlxync6zrlu33yeg5dya`
- 1 - X- `avax` `19rknw8l0grnfunjrzwxlxync6zrlu33y2jxhrg`
- 2 - X- `cascade` `19rknw8l0grnfunjrzwxlxync6zrlu33ypmtvh`
- 3 - X- `denali` `19rknw8l0grnfunjrzwxlxync6zrlu33yhc357h`
- 4 - X- `everest` `19rknw8l0grnfunjrzwxlxync6zrlu33yn44wty`
- 5 - X- `fuji` `19rknw8l0grnfunjrzwxlxync6zrlu33yxqzg0h`
- 1337 - X- `custom` `19rknw8l0grnfunjrzwxlxync6zrlu33yeg5dya`
- 12345 - X- `local` `19rknw8l0grnfunjrzwxlxync6zrlu33ynpm3qq`

Here's the mapping of `networkID` to bech32 HRP.

```
export const NetworkIDToHRP = {  
  0: "custom",  
  1: "avax",  
  2: "cascade",  
  3: "denali",  
  4: "everest",  
  5: "fuji",  
  1337: "custom",  
  12345: "local",  
}
```

Change the HRP of the bech32 address by passing in a different `networkID` when instantiating `Avalanche`.

```
// mainnet  
const networkID = 1  
const avalanche = new Avalanche(undefined, undefined, undefined, networkID)  
  
// [ 'X-avax1j2j0vzttatv73gr7j4tnd7rp4el3ngcyjy0pre' ]  
// [ 'X-avax19rknw8l0grnfunjrzwxlxync6zrlu33y2jxhrg' ]
```

```
// fuji  
const networkID = 5  
const avalanche = new Avalanche(undefined, undefined, undefined, networkID)  
  
// [ 'X-fuji1j2j0vzttatv73gr7j4tnd7rp4el3ngcy7kt70x' ]  
// [ 'X-fuji19rknw8l0grnfunjrzwxlxync6zrlu33yxqzg0h' ]
```

```
// custom  
const networkID = 1337 // also networkID = 0  
const avalanche = new Avalanche(undefined, undefined, undefined, networkID)  
  
// [ 'X-custom1j2j0vzttatv73gr7j4tnd7rp4el3ngcyp7amyv' ]  
// [ 'X-custom19rknw8l0grnfunjrzwxlxync6zrlu33yeg5dya' ]
```

```
// mainnet  
const networkID = 12345  
const avalanche = new Avalanche(undefined, undefined, undefined, networkID)  
  
// [ 'X-local1j2j0vzttatv73gr7j4tnd7rp4el3ngcyjh8q3' ]  
// [ 'X-local19rknw8l0grnfunjrzwxlxync6zrlu33ynpm3qq' ]
```

Send an Asset on the X-Chain

This example sends an asset in the X-Chain to a single recipient. The first step in this process is to create an instance of `Avalanche` connected to our `Avalanche` Platform endpoint of choice.

```
import { Avalanche, BinTools, Buffer, BN } from "avalanche"  
  
let myNetworkID = 1 //default is 3, we want to override that for our local network  
let myBlockchainID = "2oYMBNV4eNHqk2fjjV5nQLDtmNjzq5s3qs3Lo6ftnC6FBByM" // The X-Chain blockchainID on this network
```

```

let avax = new avalanche.Avalanche(
  "localhost",
  9650,
  "http",
  myNetworkID,
  myBlockchainID
)
let xchain = avax.XChain() //returns a reference to the X-Chain used by AvalancheJS

```

We're also assuming that the keystore contains a list of addresses used in this transaction.

Getting the UTXO Set

The X-Chain stores all available balances in a data store called Unspent Transaction Outputs (UTXOs). A UTXO Set is the unique list of outputs produced by transactions, addresses that can spend those outputs, and other variables such as lockout times (a timestamp after which the output can be spent) and thresholds (how many signers are required to spend the output).

For the case of this example, we're going to create a simple transaction that spends an amount of available coins and sends it to a single address without any restrictions. The management of the UTXOs will mostly be abstracted away.

However, we do need to get the UTXO Set for the addresses we're managing.

```

let myAddresses = xchain.keyChain().getAddresses() //returns an array of addresses the KeyChain manages
let addressStrings = xchain.keyChain().getAddressStrings() //returns an array of addresses the KeyChain manages as strings
let utxos = (await xchain.getUTXOs(myAddresses)).utxos

```

Spending the UTXOs

The `buildBaseTx()` helper function sends a single asset type. We have a particular assetID whose coins we want to send to a recipient address. This is an imaginary asset for this example which we believe to have 400 coins. Let's verify that we have the funds available for the transaction.

```

let assetid = "23wKfz3viWLmjWo2UZ7xWegjvnZFenGAVkouwQCeB9ubPXodG6" //avaSerialized string
let mybalance = utxos.getBalance(myAddresses, assetid) //returns 400 as a BN

```

We have 400 coins! We're going to now send 100 of those coins to our friend's address.

```

let sendAmount = new BN(100) //amounts are in BN format
let friendsAddress = "X-avax1k26jvfdzyukms95puxcceyza3lzwf5ftt0fjk" // address format is Bech32

//The below returns a UnsignedTx
//Parameters sent are (in order of appearance):
// * The UTXO Set
// * The amount being sent as a BN
// * An array of addresses to send the funds
// * An array of addresses sending the funds
// * An array of addresses any leftover funds are sent
// * The AssetID of the funds being sent
let unsignedTx = await xchain.buildBaseTx(
  utxos,
  sendAmount,
  [friendsAddress],
  addressStrings,
  addressStrings,
  assetid
)
let signedTx = unsignedTx.sign(myKeychain)
let txid = await xchain.issueTx(signedTx)

```

And the transaction is sent!

Get the Status of the Transaction

Now that we sent the transaction to the network, it takes a few seconds to determine if the transaction has gone through. We can get an updated status on the transaction using the TxID through the X-Chain.

```

// returns one of: "Accepted", "Processing", "Unknown", and "Rejected"
let status = await xchain.getTxStatus(txid)

```

The statuses can be one of Accepted, Processing, Unknown, and Rejected

- "Accepted" indicates that the transaction has been accepted as valid by the network and executed
- "Processing" indicates that the transaction is being voted on.
- "Unknown" indicates that node knows nothing about the transaction, indicating the node doesn't have it
- "Rejected" indicates the node knows about the transaction, but it conflicted with an accepted transaction

Check the Results

The transaction finally came back as `Accepted`, now let's update the UTXOSet and verify that the transaction balance is as we expected.

Note: In a real network the balance isn't guaranteed to match this scenario. Transaction fees or additional spends may vary the balance. For the purpose of this example, we assume neither of those cases.

```
let updatedUTXOs = await xchain.getUTXOs()
let newBalance = updatedUTXOs.getBalance(myAddresses, assetid)
if (newBalance.toNumber() != mybalance.sub(sendAmount).toNumber()) {
  throw Error("heyyy these should equal!")
}
```

Glacier API

The Glacier API is a performant API that allows web3 developers to more easily access the indexed blockchain data they need to build powerful applications on top of Avalanche's primary and subnetworks as well as Ethereum.

By leveraging Glacier, developers can:

- Retrieve native and ERC-20 token balances and associated pricing information
- Get details related to blocks, transactions and UTXOs
- Retrieve digital collectible (ERC-721/1155) balances and metadata
- Get native asset and token transfer history

More information about accessing Glacier can be found [here](#). If you have feedback or feature requests for the API, please submit them [here](#). Bug reports can be submitted [here](#), and any potential security issues can be reported [here](#).

Avalanche Metrics API

Power your analytics with Avalanche Metrics API such as Subnet usage, staking operations, and more. See [here for details](#).

Community Overview

Community Channels

The [Avalanche ecosystem](#) is a fast-growing network of developers collaborating to build a better future with everyone.

- Find the community on [Discord](#).
- Engage with the [Telegram](#) group and/or [Reddit](#).
- Follow Avalanche on [Twitter](#).
- Check [Avalanche's Medium posts](#).
- Watch Avalanche videos on [YouTube](#).
- Find code on [GitHub](#).
- You can reach support [here](#).

Questions? {#Questions}

We love to answer questions! Questions help everyone learn together, so please don't hesitate to ask :)

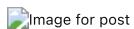
Our [Discord](#) is for any real-time conversations, from brainstorming to debugging and troubleshooting. Participants often share interesting articles and other educational content. For in-depth considerations about important decisions and projects, we suggest contributing on [GitHub](#). New issues, comments, and pull requests are welcome!

Bug Bounty

[HackenProof](#) hosts Avalanche's bug bounty program.

2021 Winners

On Aug 2, 2021, Avalanche launched a [contest](#) for the best developer tutorials to help others build on the platform. Anyone was able to submit a tutorial, with selected tutorials eligible to win a portion of the \$50,000 prize pool.



Here you can see the entries of the Avalanche Developer Contest Winners:

:::warning

These tutorials are published as a snapshot when they were written. For up-to-date information, please reach out to the owners of these projects.

:::

DeFi

- [Avalanche Chainlink tutorial](#) by [red-dev](#)
- [Pangolin Token Subgraph](#) by [Ofem Eteng](#)

Smart Contracts

- [Hardhat forking tutorial](#) by [Tristan Brunain](#)
- [Signature Verification in a dapp](#) by [red-dev](#)

Social DApps

- [eVoting dapp](#) by [Raj Ranjan](#)
- [Avalanche DAO](#) by [Arturo Castañon](#)

NFTs

- [NFT Marketplace](#) by [Berkay Saglam](#)
- [NFT tutorial](#) by [Metodi Manov](#)

Tokenization

- [Minting ERC-721s](#) by [Harpalsinh Jadeja](#)

Additional Topics

- [Chat dapp on Avalanche](#) by [Nimish Agrawal](#)
- [Auction Manager](#) by [Raj Ranjan](#)
- [Distributed File Manager](#) by [Raj Ranjan](#)

A big congratulations to the winners.

The Avalanche Foundation plans to have more contests in the future. Follow along across the community channels to learn more about future events and participate.

Please note that all the entries and the corresponding code belong to the authors and Avalanche Foundation can't take responsibility for any fitness and usage of these materials in your own projects. They're intended for educational purposes only.

Getting Started with Create React App

This project was bootstrapped with [Create React App](#).

Available Scripts

In the project directory, you can run:

```
npm start
```

Runs the app in the development mode.

Open <http://localhost:3000> to view it in the browser.

The page will reload if you make edits.

You will also see any lint errors in the console.

```
npm test
```

Launches the test runner in the interactive watch mode.

See the section about [running tests](#) for more information.

```
npm run build
```

Builds the app for production to the `build` folder.

It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.

Your app is ready to be deployed!

See the section about [deployment](#) for more information.

```
npm run eject
```

Note: this is a one-way operation. Once you `eject`, you can't go back!

If you aren't satisfied with the build tool and configuration choices, you can `eject` at any time. This command will remove the single build dependency from your project.

Instead, it will copy all the configuration files and the transitive dependencies (Webpack, Babel, ESLint, etc) right into your project so you have full control over them. All of the commands except `eject` will still work, but they will point to the copied scripts so you can tweak them. At this point you're on your own.

You don't have to ever use `eject`. The curated feature set is suitable for small and middle deployments, and you shouldn't feel obligated to use this feature. However we understand that this tool wouldn't be useful if you couldn't customize it when you are ready for it.

Learn More

You can learn more in the [Create React App documentation](#).

To learn React, check out the [React documentation](#).

Code Splitting

This section has moved here: <https://facebook.github.io/create-react-app/docs/code-splitting>

Analyzing the Bundle Size

This section has moved here: <https://facebook.github.io/create-react-app/docs/analyzing-the-bundle-size>

Making a Progressive Web App

This section has moved here: <https://facebook.github.io/create-react-app/docs/making-a-progressive-web-app>

Advanced Configuration

This section has moved here: <https://facebook.github.io/create-react-app/docs/advanced-configuration>

Deployment

This section has moved here: <https://facebook.github.io/create-react-app/docs/deployment>

npm run build Fails to Minify

This section has moved here: <https://facebook.github.io/create-react-app/docs/troubleshooting#npm-run-build-fails-to-minify>

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link
      rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
    />

    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />

    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />

    <title>NFT Marketplace</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
        This HTML file is a template.
        If you open it directly in the browser, you will see an empty page.

        You can add webfonts, meta tags, or analytics to this file.
        The build step will place the bundled scripts into the <body> tag.

        To begin the development, run `npm start` or `yarn start`.
        To create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
  <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
  <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
</html>
```

App.js

```

import React from "react";
import { ethers } from "ethers";

import AuctionArtifact from "./artifacts/Auction.json";
import AuctionManagerArtifact from "./artifacts/AuctionManager.json";
import NFTArtifact from "./artifacts/NFT.json";

const NFT_ADDRESS = "0xeb2283672cf716fF6A1d880436D3a9074Ba94375"; // NFT contract address
const AUCTIONMANAGER_ADDRESS = "0xea4b168866E439Db4A5183Dccb4951DCb5437f1E"; // AuctionManager contract address
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activeAuction: null,
      auctions: [], // Auctions to display
      bidAmount: 0, // The amount of AVAX to bid
      newAuction: {
        // newAuction is a state variable for the form
        startPrice: null,
        endTime: null,
        tokenId: null,
        minIncrement: null,
        directBuyPrice: null,
      },
      myItems: [], // Items owned by the user
    };
    this.mint = this.mint.bind(this);
    this.renderAuctionElement = this.renderAuctionElement.bind(this);
  }

  async.getItems() {
    let items = await this._nft.myItems(); // Get the tokens owned by the user
    items = items.map((x) => x.toNumber()); // Converts BigNumber to number
    this.setState({ myItems: items });
  }

  async init() {
    if (window.ethereum) {
      // if window.ethereum is defined
      await window.ethereum.enable(); // Enable the Ethereum client
      this.provider = new ethers.providers.Web3Provider(window.ethereum); // A connection to the Ethereum network
      this.signer = this.provider.getSigner(); // Holds your private key and can sign things
      this.setState({ currentAddress: await this.signer.getAddress() }); // Set the current address
      this._auctionManager = new ethers.Contract( // We will use this to interact with the AuctionManager
        AUCTIONMANAGER_ADDRESS,
        AuctionManagerArtifact.abi,
        this.signer
      );
      this._nft = new ethers.Contract(
        NFT_ADDRESS,
        NFTArtifact.abi,
        this.signer
      ); // We will use this to interact with the NFT
      this.getItems();
      this.getAuctions();
    } else {
      alert("No wallet detected"); // No wallet detected
    }
  }

  componentDidMount() {
    this.init();
  }

  async getAuctions() {
    let auctionsAddresses = await this._auctionManager.getAuctions(); // get a list of auction addresses
    let auctions = await this._auctionManager.getAuctionInfo(auctionsAddresses); // I'll just pass all the addresses here, you
    can build a pagination system if you want
    console.log(auctions);
    let new_auctions = [];

    for (let i = 0; i < auctions.endTime.length; i++) {
      let endTime = auctions.endTime[i].toNumber();
      let tokenId = auctions.tokenIds[i].toNumber();
      let auctionState = auctions.auctionState[i].toNumber();
    }
  }
}

```

```

let startPrice = ethers.utils.formatEther(auctions.startPrice[i]);
let directBuyPrice = ethers.utils.formatEther(auctions.directBuy[i]);
let highestBid = ethers.utils.formatEther(auctions.highestBid[i]);

let owner = auctions.owner[i];

let newAuction = {
  endTime: endTime,
  startPrice: startPrice,
  owner: owner,
  directBuyPrice: directBuyPrice,
  tokenId: tokenId,
  highestBid: highestBid,
  auctionState: auctionState,
  auctionAddress: auctionsAddresses[i],
};
new_auctions.push(newAuction);
}

this.setState({ auctions: new_auctions }); // Update the state
}

async createAuction() {
  if (
    !this.state.newAuction.minIncrement ||
    !this.state.newAuction.directBuyPrice ||
    !this.state.newAuction.startPrice ||
    !this.state.newAuction.endTime ||
    !this.state.newAuction.tokenId
  )
    return alert("Fill all the fields");

  let { hash: allowance_hash } = await this._nft.approve(
    AUCTIONMANAGER_ADDRESS,
    this.state.newAuction.tokenId
  ); // Approve the AUCTIONMANAGER to transfer the token
  console.log("Approve Transaction sent! Hash:", allowance_hash);
  await this.provider.waitForTransaction(allowance_hash); // Wait till the transaction is mined
  console.log("Transaction mined!");

  let { hash } = await this._auctionManager.createAuction(
    // Create an auction
    this.state.newAuction.endTime * 60, // Converting minutes to seconds
    ethers.utils.parseEther(this.state.newAuction.minIncrement.toString()), // Minimum increment in AVAX
    ethers.utils.parseEther(this.state.newAuction.directBuyPrice.toString()), // Direct buy price in AVAX
    ethers.utils.parseEther(this.state.newAuction.startPrice.toString()), // Start price in AVAX
    NFT_ADDRESS, // The address of the NFT token
    this.state.newAuction.tokenId // The id of the token
  );
  console.log("Transaction sent! Hash:", hash);
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  console.log("Transaction mined!");
  alert(`Transaction Mined! Hash: ${hash}`);
}

async mint() {
  // hash is the hash of the transaction
  let { hash } = await this._nft.getItem({
    // Calling the getItem function of the contract
    value: ethers.utils.parseEther("0.5"), // 0.5 AVAX
  });
  console.log("Transaction sent! Hash:", hash);
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  console.log("Transaction mined!");
  alert(`Transaction sent! Hash: ${hash}`);
}

renderAuctionElement(auction) {
  let state = "";
  if (auction.auctionState === 0) {
    state = "Open";
  }
  if (auction.auctionState === 1) {
    state = "Cancelled";
  }
  if (auction.auctionState === 2) {
    state = "Ended";
  }
}

```

```

        }

        if (auction.auctionState === 3) {
            state = "Direct Buy";
        }

        return (
            <div style={{ background: "yellow" }} class="col">
                <p>ID: {auction.tokenId}</p> /* ID of the token */
                <p>Highest Bid: {auction.highestBid || 0}</p>
                /* Highest bid */

                <p>Direct Buy: {auction.directBuyPrice}</p> /* Direct buy price */
                <p>Starting Price: {auction.startPrice}</p> /* Starting price */
                <p>Owner: {auction.owner}</p> /* Owner of the token */
                <p>
                    /* Convert timestamp to minutes */
                    End Time:{" "}
                    {Math.round((auction.endTime * 1000 - Date.now()) / 1000 / 60)}{" "}
                
                /* Time left in minutes */
                minutes
            </p>
            <p>Auction State: {state}</p>
            <button
                class="btn-primary"
                onClick={() => this.setActiveAuction(auction)}
            >
                See More
            </button>
        </div>
    );
}

async placeBid(amount) {
    if (!amount) return;
    amount = ethers.utils.parseEther(amount.toString()); // Amount in AVAX
    let { hash } = await this._auction.placeBid({ value: amount }); // Place a bid
    await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
    alert(`Transaction sent! Hash: ${hash}`); // Show the transaction hash
    this.setActiveAuction(this.state.activeAuction); // Update the active auction
}

async cancelAuction() {
    let { hash } = await this._auction.cancelAuction(); // Cancel the auction
    await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
    alert(`Auction Canceled! Hash: ${hash}`); // Show the transaction hash
    window.location.reload(); // Reload the page
}

async withdrawFunds() {
    let { hash } = await this._auction.withdrawFunds(); // Withdraw the funds
    await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
    alert(`Withdrawal Successful! Hash: ${hash}`); // Show the transaction hash
    window.location.reload(); // Reload the page
}

async withdrawToken() {
    let { hash } = await this._auction.withdrawToken(); // Withdraw the NFT token
    await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
    alert(`Withdrawal Successful! Hash: ${hash}`); // Show the transaction hash
    window.location.reload(); // Reload the page
}

async setActiveAuction(auction) {
    this._auction = new ethers.Contract( // Create a new instance of the contract
        auction.auctionAddress,
        AuctionArtifact.abi,
        this.signer
    );

    let previousBids = await this._auction.allBids(); // Get the bids
    let bids = []; // A list of bids
    for (let i = 0; i < previousBids[0].length; i++) {
        // Loop through the bids
        bids.push({
            // Add the bid to the list
            bidder: previousBids[0][i], // The bidder
            bid: ethers.utils.formatEther(previousBids[1][i]), // The bid
        });
    }
}

```

```

auction.bids = bids; // Add the bids array to the auction object

let auctionTokenValue = await this._nft.itemValue(auction.tokenId); // Get the value of the token
auctionTokenValue = auctionTokenValue.toNumber(); // Convert BigNumber to number
auction.auctionTokenValue = auctionTokenValue; // Add the value of the token to the auction object

let highestBidder = await this._auction.maxBidder(); // Get the highest bidder
auction.highestBidder = highestBidder; // Add the highest bidder to the auction object

let minIncrement = await this._auction.minIncrement(); // Get the minimum increment
auction.minIncrement = ethers.utils.formatEther(minIncrement); // Add the minimum increment to the auction object

this.setState({ activeAuction: auction }); // Update the state
}

renderActiveAuction() {
  let activeAuction = this.state.activeAuction;
  let state = "";
  if (activeAuction.auctionState === 0) {
    // If the auction is open
    state = "Open";
  }
  if (activeAuction.auctionState === 1) {
    // If the auction is cancelled
    state = "Cancelled";
  }
  if (activeAuction.auctionState === 2) {
    // If the auction is ended
    state = "Ended";
  }
  if (activeAuction.auctionState === 3) {
    // If the auction is ended by a direct buy
    state = "Direct Buy";
  }
  let isOwner =
    this.state.currentAddress.toString().toLowerCase() ===
    activeAuction.owner.toString().toLowerCase(); // Check if the current address is the owner
  let isAuctionOpen = state === "Open"; // Check if the auction is open
  let isAuctionCancelled = state === "Cancelled"; // Check if the auction is cancelled
  let isAuctionEnded = state === "Ended" || state === "Direct Buy"; // Check if the auction is ended
  let isHighestBidder =
    this.state.currentAddress === activeAuction.highestBidder; // Check if the current address is the highest bidder

  return (
    <div>
      <div class="col">
        <button
          class="btn-secondary"
          onClick={() => this.setState({ activeAuction: null })}
        >
          Go Back
        </button>
        <p>ID: {activeAuction.tokenId}</p> /* ID of the token */
        <p>Highest Bid: {activeAuction.highestBid || 0} AVAX</p>
        /* Highest bid */
        <p>Direct Buy: {activeAuction.directBuyPrice} AVAX</p>{" "}
        /* Direct buy price */
        <p>Minimum Increment: {activeAuction.minIncrement} AVAX</p>{" "}
        /* Minimum increment in AVAX */
        <p>Starting Price: {activeAuction.startPrice} AVAX</p>{" "}
        /* Starting price */
        <p>Owner: {activeAuction.owner}</p> /* Owner of the token */
        <p>
          End Time:{" "}
          {Math.round(
            (activeAuction.endTime * 1000 - Date.now()) / 1000 / 60
          )}" "
        /* Time left in minutes */
        minutes
      </p>
      <p>Auction State: {state}</p>
      <p>Token Value: {activeAuction.auctionTokenValue}</p>
    </div>
    <div class="col">
      <h3>Bids</h3>
      <table class="table">
        <thead>
          <tr>

```

```

        <th>Bidder</th>
        <th>Bid</th>
    </tr>
</thead>
<tbody>
{activeAuction.bids.map((bid) => {
    return (
        <tr key={bid.bidder}>
            <td>{bid.bidder}</td>
            <td>{bid.bid} AVAX</td>
        </tr>
    );
})
</tbody>
</table>
</div>
<div class="col">
{isAuctionOpen && !isOwner ? (
    <div>
        <input
            type="number"
            placeholder="0.5"
            onChange={(e) => this.setState({ bidAmount: e.target.value })}
        />
        <button
            class="btn-primary btn"
            onClick={() => this.placeBid(this.state.bidAmount)}
        >
            Place Bid
        </button>
    </div>
) : null}
{isOwner && isAuctionOpen && activeAuction.bids.length == 0 ? (
    <button onClick={() => this.cancelAuction()} class="btn-danger btn">
        Cancel Auction
    </button>
) : null}

{isOwner && isAuctionEnded && activeAuction.bids.length > 0 ? (
    <button
        onClick={() => this.withdrawFunds()}
        class="btn-secondary btn"
    >
        Withdraw Funds
    </button>
) : null}
{((activeAuction.bids.length == 0 && isOwner) || isHighestBidder) &&
isAuctionEnded ? (
    <button
        onClick={() => this.withdrawToken()}
        class="btn-secondary btn"
    >
        Withdraw Token
    </button>
) : null}
</div>
</div>
);
}

render() {
    return (
        <div>
            <div class="jumbotron d-flex align-items-center">
                <div class="container">
                    {this.state.activeAuction != null ? (
                        this.renderActiveAuction()
                    ) : (
                        <div class="auctions row">
                            {this.state.auctions.map(this.renderAuctionElement)}
                        </div>
                    )
                )
            </div>
        </div>
        <div class="container">
            <form>
                <div class="mb-3">
                    <label for="startprice" class="form-label">

```

```

        Start Price
    </label>
    <input
        value={this.state.newAuction.startPrice}
        onChange={(e) =>
            this.setState({
                newAuction: {
                    ...this.state.newAuction,
                    startPrice: e.target.value,
                },
            })
        }
        type="number"
        class="form-control"
        id="startprice"
    />
    <label for="startprice" class="form-label">
        Token Id
    </label>
    <input
        value={this.state.newAuction tokenId}
        onChange={(e) =>
            this.setState({
                newAuction: {
                    ...this.state.newAuction,
                    tokenId: e.target.value,
                },
            })
        }
        type="number"
        class="form-control"
        id="startprice"
    />
    <label class="form-label">Minimum Increment</label>
    <input
        value={this.state.newAuction.minIncrement}
        onChange={(e) =>
            this.setState({
                newAuction: {
                    ...this.state.newAuction,
                    minIncrement: e.target.value,
                },
            })
        }
        type="number"
        class="form-control"
    />
    <label class="form-label">Direct Buy Price</label>
    <input
        value={this.state.newAuction.directBuyPrice}
        onChange={(e) =>
            this.setState({
                newAuction: {
                    ...this.state.newAuction,
                    directBuyPrice: e.target.value,
                },
            })
        }
        type="number"
        class="form-control"
    />

    <label class="form-label">Duration In Minutes</label>
    <input
        value={this.state.newAuction.endTime}
        onChange={(e) =>
            this.setState({
                newAuction: {
                    ...this.state.newAuction,
                    endTime: e.target.value,
                },
            })
        }
        type="number"
        class="form-control"
    />
</div>
```

```

        <button
          type="button"
          onClick={() => this.createAuction()}
          class="btn btn-primary"
        >
          Create Auction
        </button>
        <button
          type="button"
          onClick={() => this.mint()}
          class="btn btn-danger"
        >
          Mint NFT
        </button>
        <p>
          Your items
          <br />
          {(this.state.myItems || []).map((x) => `id: ${x} `) || ""}
        </p>
      </form>
    </div>
  );
}

export default App;

```

Index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './app';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();

```

ReportWebVitals.js

```

const reportWebVitals = onPerfEntry => {
  if (onPerfEntry && onPerfEntry instanceof Function) {
    import('web-vitals').then(({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
      getCLS(onPerfEntry);
      getFID(onPerfEntry);
      getFCP(onPerfEntry);
      getLCP(onPerfEntry);
      getTTFB(onPerfEntry);
    });
  }
};

export default reportWebVitals;

```

SetupTests.js

```

// jest-dom adds custom jest matchers for asserting on DOM nodes.
// allows you to do things like:
// expect(element).toHaveTextContent(/react/i)

```

```
// learn more: https://github.com/testing-library/jest-dom
import '@testing-library/jest-dom';
```

NFT Marketplace on Avalanche

Step by step tutorial to build your own NFT marketplace on Avalanche using Hardhat and React.

Table of Contents

- [Introduction](#)
 - [Prerequisites](#)
 - [Requirements](#)
- [Getting Started](#)
 - [Building the workspace](#)
 - [Creating the NFT Token](#)
- [Contracts for the marketplace](#)
 - [Auction Contract](#)
 - [Starting with the functions](#)
 - [Events](#)
 - [Storage variables](#)
 - [Constructor function](#)
 - [Building the functions](#)
 - [Get the auction state](#)
 - [Placing bids](#)
 - [Cancel auction](#)
 - [Withdraw Token](#)
 - [Withdraw Funds](#)
 - [Get all bids](#)
 - [Auction Manager Contract](#)
 - [Building the functions](#)
 - [Create auction](#)
 - [Get Auctions](#)
- [AVAX Fuji Testnet](#)
- [Deploying the contracts](#)
- [React App](#)
 - [Building the form](#)
 - [Interacting with contracts](#)
 - [Contract ABIs](#)
 - [Detecting and connecting to MetaMask](#)
 - [Minting an NFT](#)
 - [Owned NFT's](#)
 - [Creating an auction](#)
 - [Getting the list of auctions](#)
 - [Building the auction page](#)
 - [Rendering the selected auction](#)
 - [Auction Functions](#)
 - [Placing Bids](#)
 - [Withdraw Token](#)
 - [Withdraw Funds](#)
 - [Cancel Auction](#)
 - [Polishing the buttons](#)
 - [Place Bid Area](#)
 - [Cancel Auction Button](#)
 - [Withdraw Funds Button](#)
 - [Withdraw Token Button](#)
 - [Finishing the React App](#)
- [Using with other NFT contracts](#)
- [Next steps](#)
 - [Tests](#)
 - [Things to consider](#)
- [Deploying to the Avalanche Mainnet](#)
- [Conclusion](#)

Introduction

We will be using [React JS](#) to build the frontend of our NFT Marketplace and we will use [Ethers](#) to interact with the smart contracts on the frontend. For the backend, we will be using our smart contracts that will be deployed on the [Avalanche](#) chain and that's all. Our NFT Marketplace will live on the blockchain. Totally decentralized! We will be using Hardhat to compile and deploy our smart contracts.

Prerequisites

- Basic knowledge of [React JS](#).

- React is a JavaScript library for building user interfaces. React makes it easier to create interactive UIs. The whole UI of the NFT Marketplace is just a single file. You may use anything you would like to build your UI (Angular, Vanilla JS, Vue.js). The important part is to know how to interact with the blockchain and our smart contracts.
- Very basic knowledge of [Hardhat](#).
 - Hardhat is a development environment to compile, deploy, test, and debug your solidity code. You may use [Truffle](#) or any other framework that lets you deploy smart contracts as well. I do prefer using Hardhat.
- Basic knowledge of [Ethers](#) library.
 - [Ethers](#) is a JavaScript library for interacting with the blockchain and our smart contracts. Once you get familiar with it, you may use it to interact with the chain in any JavaScript project. You may also use [Web3 JS](#) if you are more familiar with it. I find it easier to work with Ethers.
- Basic knowledge of [Solidity](#) language.
 - We will be writing our smart contracts in **solidity**. A piece of basic knowledge is required for you to understand the concepts fully.

Requirements

- [Node JS](#) and [npm](#) must be installed.
- [Hardhat](#) must be installed.
- [MetaMask](#) extension must be installed on your browser.
- [create-react-app](#) must be installed.

Getting Started

Building the Workspace

Let's get started by setting up our workspace using [Hardhat](#).

- Execute `$ npx hardhat init` in your working directory.
- When you are prompted on the terminal
 - Choose Create a basic sample project .
 - Add the `.gitignore` file.
 - Install the dependencies.
- Delete the `Greeter.sol` file inside the contracts folder.

This will set up our initial workspace.

Creating the NFT Token

First, we need to create an NFT token that will be displayed in our marketplace.

Create a simple ERC-721 token.

[NFT.sol](#)

```
contract NFT is ERC721 {
  using Counters for Counters.Counter;
  Counters.Counter private _tokenIds;
  mapping (uint => uint) public itemValue;
  uint maxValue = 10000; // Max value of an item

  constructor() ERC721("Super NFT", "SPRNFT") {}

  /**
   * Returns a random number
   */
  function random() private view returns (uint) {
    return uint(keccak256(abi.encodePacked(block.difficulty, block.timestamp, block.number)));
  }

  function myItems() external view returns (uint[] memory items) {
    // Returns an array of items that the user owns
    items = new uint[](balanceOf(msg.sender));
    uint _counter = 0;
    for(uint i = 1; i < _tokenIds.current(); i++) { // i = 1 because the token counter is increased before the id is assigned
      if(ownerOf(i) == msg.sender) { // if the user owns the item
        items[_counter] = i; // add the item to the array
        _counter++; // increase the counter
      }
    }
    return items;
  }

  function getItem() payable public returns (uint256) {
    require(msg.value == 0.5 ether); // 0.5 AVAX is the cost of an item
    _tokenIds.increment(); // Increase the counter
    uint256 newItemId = _tokenIds.current(); // Get the current counter value
    _mint(msg.sender, newItemId); // Mint the new item to the user
    itemValue[newItemId] = random() % maxValue; // Set the item value to a random number modulus used to make sure that the value isn't bigger than maxValue
    return newItemId; // Return the new item id
  }
}
```

```
    }
}
```

ALERT

Secure random-number generation in the blockchain is a very difficult problem. Our method here is insecure, but since this is just an example, it will be good enough for our purposes.

ALERT

The logic here is very simple.

- User calls the `getItem()` function and pays 0.5 AVAX
- The value of the token is determined using a random number.
- The token is minted to the user.

And that's all.

Contracts for the Marketplace

We will have two contracts, `AuctionManager` and `Auction` contracts. `AuctionManager` contract will create the auctions by deploying the `Auction` contract using the parameters. Here is how it will work:

- The auction creator enters parameters for direct buy price, starting price, minimum bid increment, the token id, and the auction end time.
- The auction creator gives approval for the NFT that will be auctioned to the `AuctionManager`.
- The `AuctionManager` creates an `Auction` with the parameters given and then transfers the NFT to the newly created `Auction` contract.
- After the auction is completed, the auction creator and the auction winner must call the withdrawal functions to retrieve their tokens.

Auction Contract

Starting with the Functions

Our auction contract will allow users to place bids, cancel the auction if there are no bids and withdraw their tokens and funds after the auction has been completed. We will also need a function to get the current state of the auction.

```
contract Auction {
    function placeBid() payable external returns(bool) {} // Place a bid on the auction
    function withdrawToken() external returns(bool) {} // Withdraw the token after the auction is over
    function withdrawFunds() external returns(bool) {} // Withdraw the funds after the auction is over
    function cancelAuction() external returns(bool) {} // Cancel the auction
    function getAuctionState() public view returns(uint) {} // Get the auction state
    function allBids() external view returns (address[] memory, uint256[] memory) {} // Returns a list of all bids and addresses
}
```

Events

There will be four events in the contract. `NewBid` will be emitted each time there is a new bid. `WithdrawToken` will be emitted when the highest bidder withdraws the token. `WithdrawFunds` will be emitted when the auction owner withdraws the funds. `AuctionCancelled` will be emitted when the auction gets canceled.

```
contract Auction {
    event NewBid(address bidder, uint bid); // A new bid was placed
    event WithdrawToken(address withdrawer); // The auction winner withdrew the token
    event WithdrawFunds(address withdrawer, uint256 amount); // The auction owner withdrew the funds
    event AuctionCancelled(); // The auction was cancelled
}
```

Storage Variables

We will need to declare a few variables.

- `endTime` is the end timestamp of the auction. The auction will end if the current timestamp is equal or greater than the `endTime`.
- `startTime` is the timestamp that marks the start of the auction.
- `maxBid` the amount of max bid.
- `maxBidder` is the address of the max bidder.
- `creator` is the address of the auction creator.
- `bids` array of bids, we will use this to display recent bids on the web page.
- `tokenId` is the id of the token that has been auctioned.
- `isCancelled` will be true if the auction has been canceled.
- `isDirectBuy` will be true if someone placed a bid with a higher or equal value than the direct buy price.
- `minIncrement` is the minimum increment for the bid.
- `directBuyPrice` is the price for a direct buy.
- `startPrice` is the starting price for the auction.
- `nftAddress` is the address of the NFT contract
- `_nft` is the NFT token.

```
contract Auction {
    uint256 public endTime; // Timestamp of the end of the auction (in seconds)
```

```

uint256 public startTime; // The block timestamp which marks the start of the auction
uint maxBid; // The maximum bid
address maxBidder; // The address of the maximum bidder
address creator; // The address of the auction creator
Bid[] public bids; // The bids made by the bidders
uint tokenId; // The id of the token
bool isCancelled; // If the auction is cancelled
bool isDirectBuy; // True if the auction ended due to direct buy
uint minIncrement; // The minimum increment for the bid
uint directBuyPrice; // The price for a direct buy
uint startPrice; // The starting price for the auction
IERC721 _nft; // The NFT token

enum AuctionState {
    OPEN,
    CANCELLED,
    ENDED,
    DIRECT_BUY
}

struct Bid { // A bid on an auction
    address sender;
    uint256 bid;
}
}
}

```

We would be using block numbers instead of timestamps. If we were developing this application on another chain such as Ethereum Mainnet, block timestamps are being set by the miners, and they are spoofable. However, in Avalanche, there is no set block rate; thus, we cannot rely on block numbers. We will be using `block.timestamp` to measure time. Timestamps are guaranteed to be within 30 seconds of real-time therefore we do not need to worry about it. [Read more about it here.](#)

The auction will end when the current timestamp is greater than or equal to the `endTime`. `endTime` is the end timestamp of the auction.

Constructor Function

We will need to get some parameters from the `AuctionManager` contract.

```

constructor(address _creator,uint _endTime,uint _minIncrement,uint _directBuyPrice, uint _startPrice,address _nftAddress,uint
_tokenId) {
    creator = _creator; // The address of the auction creator
    endTime = block.timestamp + _endTime; // The timestamp which marks the end of the auction (now + 30 days = 30 days from
now)
    startBlock = block.timestamp; // The timestamp which marks the start of the auction
    minIncrement = _minIncrement; // The minimum increment for the bid
    directBuyPrice = _directBuyPrice; // The price for a direct buy
    startPrice = _startPrice; // The starting price for the auction
    _nft = IERC721(_nftAddress); // The address of the nft token
    nftAddress = _nftAddress;
    tokenId = _tokenId; // The id of the token
    maxBidder = _creator; // Setting the maxBidder to auction creator.
}

```

Building the Functions

Get the Auction State

Let's start with the `getAuctionState` function as we are going to use it in other functions. First, we need to check if the auction has been canceled and return `AuctionState.CANCELLED` in that case. Then, we need to check if anyone bid more or equal to the direct buy price and return `AuctionState.DIRECT_BUY` in that case. Then, we need to check if the current timestamp is greater or equal to the `endTime` and return `AuctionState.ENDED` in that case. Otherwise, we will return `AuctionState.OPEN`.

```

// Get the auction state
function getAuctionState() public view returns(AuctionState) {
    if(isCancelled) return AuctionState.CANCELLED; // If the auction is cancelled return CANCELLED
    if(isDirectBuy) return AuctionState.DIRECT_BUY; // If the auction is ended by a direct buy return DIRECT_BUY
    if(block.timestamp >= endTime) return AuctionState.ENDED; // The auction is over if the block timestamp is greater than the
end timestamp, return ENDED
    return AuctionState.OPEN; // Otherwise return OPEN
}

```

Placing Bids

Let's start building the function for placing bids. The user calls the `placeBid` function and sends an amount of AVAX.

Firstly, there are few things we need to check;

- The bidder cannot be the auction creator
- The auction must be open.
- The bid must be greater than the start price.

- The bid must be greater than the highest bid + minimum bid increment.

```
function placeBid() payable external returns(bool){
    require(msg.sender != creator); // The auction creator can not place a bid
    require(getAuctionState() == AuctionState.OPEN); // The auction must be open
    require(msg.value > startPrice); // The bid must be higher than the starting price
    require(msg.value > maxBid + minIncrement); // The bid must be higher than the current bid
}
```

Next, we will set the new max bid and the max bidder. After that, we will store the value of the last highest bid and the bidder because we will need this information later.

```
address lastHighestBidder = maxBidder; // The address of the last highest bidder
uint256 lastHighestBid = maxBid; // The last highest bid
maxBid = msg.value; // The new highest bid
maxBidder = msg.sender; // The address of the new highest bidder
```

Next, we have to check if the bid (`msg.value`) is greater or equal to the direct buy price. In that case, we will need to set the `isDirectBuy` to `true`; thus, close the auction.

```
if(msg.value >= directBuyPrice){ // If the bid is higher or equal to the direct buy price
    isDirectBuy = true; // The auction has ended
}
```

Next, we have to push the value of the new bid into the bids array.

```
bids.push(Bid(msg.sender,msg.value));
```

Finally, if there is a previous bid, we have to refund the previous highest bid to the previous bidder. Also, we should emit a `NewBid` event.

```
if(lastHighestBid != 0){ // if there is a bid
    address(uint160(lastHighestBidder)).transfer(lastHighestBid); // refund the previous bid to the previous highest bidder
}

emit NewBid(msg.sender,msg.value); // emit a new bid event

return true;
```

Here is the complete function

```
// Place a bid on the auction
function placeBid() payable external returns(bool){
    require(msg.sender != creator); // The auction creator can not place a bid
    require(getAuctionState() == AuctionState.OPEN); // The auction must be open
    require(msg.value > startPrice); // The bid must be higher than the starting price
    require(msg.value > maxBid + minIncrement); // The bid must be higher than the current bid + the minimum increment

    address lastHighestBidder = maxBidder; // The address of the last highest bidder
    uint256 lastHighestBid = maxBid; // The last highest bid
    maxBid = msg.value; // The new highest bid
    maxBidder = msg.sender; // The address of the new highest bidder
    if(msg.value >= directBuyPrice){ // If the bid is higher than the direct buy price
        isDirectBuy = true; // The auction has ended
    }
    bids.push(Bid(msg.sender,msg.value)); // Add the new bid to the list of bids

    if(lastHighestBid != 0){ // if there is a bid
        address(uint160(lastHighestBidder)).transfer(lastHighestBid); // refund the previous bid to the previous highest bidder
    }

    emit NewBid(msg.sender,msg.value); // emit a new bid event
    return true; // The bid was placed successfully
}
```

Cancel Auction

This is a very simple function. If there are no bids and the auction is open, the auction creator should be able to cancel the auction.

```
function cancelAuction() external returns(bool){ // Cancel the auction
    require(msg.sender == creator); // Only the auction creator can cancel the auction
    require(getAuctionState() == AuctionState.OPEN); // The auction must be open
    require(maxBid == 0); // The auction must not be cancelled if there is a bid
    isCancelled = true; // The auction has been cancelled
    _nft.transferFrom(address(this), creator, tokenId); // Transfer the NFT token to the auction creator
    emit AuctionCanceled(); // Emit Auction Canceled event
```

```

    return true;
}

```

Withdraw Token

After the auction has been completed, the highest bidder should be able to withdraw the NFT token. The `msg.sender` must be the highest bidder and the auction must be completed by either direct buy or timeout; otherwise the function must revert. After that, The NFT with the token id `tokenId` will be transferred to the highest bidder.

We've set the max bidder initial value to the auction creator's wallet address, so if the auction times out and no one bids, the auction creator will be able to withdraw the token.

```

// Withdraw the token after the auction is over
function withdrawToken() external returns(bool){
    require(getAuctionState() == AuctionState.ENDED || getAuctionState() == AuctionState.DIRECT_BUY); // The auction must be
ended by either a direct buy or timeout
    require(msg.sender == maxBidder); // The highest bidder can only withdraw the token
    _nft.transferFrom(address(this), maxBidder, tokenId); // Transfer the token to the highest bidder
    emit WithdrawToken(maxBidder); // Emit a withdraw token event
}

```

Withdraw Funds

After the auction has been completed, the auction creator should be able to withdraw the funds. The `msg.sender` must be the auction owner and the auction must be completed by either a direct buy or timeout; otherwise, the function must revert. After that, the max bid amount must be transferred to the auction creator.

```

// Withdraw the funds after the auction is over
function withdrawFunds() external returns(bool){
    require(getAuctionState() == AuctionState.ENDED || getAuctionState() == AuctionState.DIRECT_BUY); // The auction must be
ended by either a direct buy or timeout
    require(msg.sender == creator); // The auction creator can only withdraw the funds
    address(uint160(msg.sender)).transfer(maxBid);
    emit WithdrawFunds(msg.sender,maxBid); // Emit a withdraw funds event
}

```

Get All Bids

We will need a function to get a list of all bids and the bidders.

```

// Returns a list of all bids and addresses
function allBids()
external
view
returns (address[] memory, uint256[] memory)
{
    address[] memory addrs = new address[](bids.length);
    uint256[] memory bidPrice = new uint256[](bids.length);
    for (uint256 i = 0; i < bids.length; i++) {
        addrs[i] = bids[i].sender;
        bidPrice[i] = bids[i].bid;
    }
    return (addrs, bidPrice);
}

```

Our Auction contract is ready. Here is the complete code: [Auction Contract](#)

Auction Manager Contract

We will use this contract to get a list of all auctions, and to create new ones. Let's start with the basic structure.

```

pragma solidity ^0.7.0;

import "./Auction.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";

contract AuctionManager {
    uint _auctionIdCounter; // auction Id counter
    mapping(uint => Auction) public auctions; // auctions

    function createAuction(uint _endTime, uint _minIncrement, uint _directBuyPrice,uint _startPrice,address _nftAddress,uint
_tokenId) external returns (bool) {} // create an auction
    function getAuctions() external view returns(address[] memory _auctions) {} // Return a list of all auctions
    function getAuctionInfo(address[] calldata _auctionsList) external view
    returns (
        uint256[] memory directBuy,
        address[] memory owner,
        uint256[] memory highestBid,
    )
}

```

```

        uint256[] memory tokenIds,
        uint256[] memory endTime,
        uint256[] memory startPrice,
        uint256[] memory auctionState
    ) {} // Return the information of each auction address
}

```

We will need two functions, one for creating an auction and the other one for getting a list of auctions. We will use the counter to assign a unique id for each auction so we can keep track of them.

Building the Functions

Create Auction

This is a very straightforward function.

First, we need to check that:

- Direct buy price is greater than zero.
- Start price is greater than the direct buy price.
- End time is greater than 5 minutes, so no one will be able to create an auction that lasts few seconds. That wouldn't make sense.

```

require(_directBuyPrice > 0); // direct buy price must be greater than 0
require(_startPrice < _directBuyPrice); // start price is smaller than direct buy price
require(_endTime > 5 minutes); // end time must be greater than 5 minutes (setting it to 5 minutes)

```

Then, we will create a new Auction contract using the parameters and assign an id to it.

```

uint auctionId = _auctionIdCounter; // get the current value of the counter
_auctionIdCounter++; // increment the counter
Auction auction = new Auction(msg.sender, _endTime, _minIncrement, _directBuyPrice, _startPrice, _nftAddress, _tokenId); // create the auction

```

Finally, we will transfer the NFT token to the newly generated Auction contract and update our auctions map.

```

IERC721 _nftToken = IERC721(_nftAddress); // get the nft token
_nftToken.transferFrom(msg.sender, address(auction), _tokenId); // transfer the token to the auction
auctions[auctionId] = auction; // add the auction to the map
return true;

```

See the complete function.

```

// create an auction
function createAuction(uint _endTime, uint _minIncrement, uint _directBuyPrice,uint _startPrice,address _nftAddress,uint _tokenId) external returns (bool){
    require(_directBuyPrice > 0); // direct buy price must be greater than 0
    require(_startPrice < _directBuyPrice); // start price is smaller than direct buy price
    require(_endTime > 5 minutes); // end time must be greater than 5 minutes (setting it to 5 minutes for testing you can set it to 1 days or anything you would like)

    uint auctionId = _auctionIdCounter; // get the current value of the counter
    _auctionIdCounter++; // increment the counter
    Auction auction = new Auction(msg.sender, _endTime, _minIncrement, _directBuyPrice, _startPrice, _nftAddress, _tokenId); // create the auction
    IERC721 _nftToken = IERC721(_nftAddress); // get the nft token
    _nftToken.transferFrom(msg.sender, address(auction), _tokenId); // transfer the token to the auction
    auctions[auctionId] = auction; // add the auction to the map
    return true;
}

```

Get Auctions

This function will iterate over all auctions and return a list of auction addresses.

```

// Return a list of all auctions
function getAuctions() external view returns(address[] memory _auctions) {
    _auctions = new address[](_auctionIdCounter); // create an array of size equal to the current value of the counter
    for(uint i = 0; i < _auctionIdCounter; i++) { // for each auction
        _auctions[i] = address(auctions[i]); // add the address of the auction to the array
    }
    return _auctions; // return the array
}

```

Then, we can use this list of auctions to display them on the web page. We will need a function to obtain information about the auctions. Given an array of auction addresses, we would like to get direct buy price, auction creator, starting price, highest bid, token id, auction state, and the end time.

```

// Return the information of each auction address
function getAuctionInfo(address[] calldata _auctionsList)
    external
    view
    returns (
        uint256[] memory directBuy,
        address[] memory owner,
        uint256[] memory highestBid,
        uint256[] memory tokenIds,
        uint256[] memory endTime,
        uint256[] memory startPrice,
        uint256[] memory auctionState
    )
{
    directBuy = new uint256[](_auctionsList.length); // create an array of size equal to the length of the passed array
    owner = new address[](_auctionsList.length); // create an array of size equal to the length of the passed array
    highestBid = new uint256[](_auctionsList.length);
    tokenIds = new uint256[](_auctionsList.length);
    endTime = new uint256[](_auctionsList.length);
    startPrice = new uint256[](_auctionsList.length);
    auctionState = new uint256[](_auctionsList.length);

    for (uint256 i = 0; i < _auctionsList.length; i++) { // for each auction
        directBuy[i] = Auction(auctions[i]).directBuyPrice(); // get the direct buy price
        owner[i] = Auction(auctions[i]).creator(); // get the owner of the auction
        highestBid[i] = Auction(auctions[i]).maxBid(); // get the highest bid
        tokenIds[i] = Auction(auctions[i]).tokenId(); // get the token id
        endTime[i] = Auction(auctions[i]).endTime(); // get the end time
        startPrice[i] = Auction(auctions[i]).startPrice(); // get the start price
        auctionState[i] = uint(Auction(auctions[i]).getAuctionState()); // get the auction state
    }

    return ( // return the arrays
        directBuy,
        owner,
        highestBid,
        tokenIds,
        endTime,
        startPrice,
        auctionState
    );
}

```

Here is the complete code: [AuctionManager.sol](#)

That's all for the contracts!

AVAX Fuji Testnet

We are going to test the marketplace on AVAX Fuji Testnet. First, you need to add AVAX Fuji Testnet to MetaMask. Open MetaMask and view networks, then click on `Custom RPC`.



We will deploy our contracts on Fuji Testnet. Fuji Testnet Settings:

- Network Name: Avalanche Fuji C-Chain
- New RPC URL: <https://api.avax-test.network/ext/bc/C/rpc>
- ChainID: 43113
- Symbol: AVAX
- Explorer: <https://testnet.snowtrace.io>

Next, we will add the network configuration in hardhat config file `hardhat.config.js`. If you do not know how that file works then take a look at [here](#).

```

networks: {
    ...
    fuji: {
        url: "https://api.avax-test.network/ext/bc/C/rpc",
        chainId: 43113,
        accounts: [
            "PRIVATE_KEY",
        ],
    },
}

```

Lastly, we will need some AVAX to deploy our contracts. Use this [AVAX Fuji Testnet Faucet](#) to get some test AVAX for free.

Deploying the Contracts

We need to deploy our `NFT` and `AuctionManager` contracts to the Fuji Testnet. We will use hardhat to deploy the contracts, [learn more about it here](#). Start by editing the `scripts/deploy.js` file.

```
const main = async () => {
  const NftToken = await ethers.getContractFactory("NFT"); // NFT token contract
  const nftToken = await NftToken.deploy(); // NFT token contract instance
  await nftToken.deployed(); // wait for contract to be deployed
  const nftTokenAddress = await nftToken.address; // NFT token contract address

  const AuctionManager = await ethers.getContractFactory("AuctionManager"); // Auction Manager contract
  const auctionManager = await AuctionManager.deploy(); // Auction Manager contract instance
  await auctionManager.deployed(); // wait for contract to be deployed
  const auctionManagerAddress = await auctionManager.address; // Auction Manager contract address

  console.log(`NFT deployed to: ${nftTokenAddress}`); // NFT token contract address
  console.log(`Auction Manager deployed to: ${auctionManagerAddress}`); // Auction Manager contract address
};

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
});
```

After you are done editing `deploy.js`, execute the following lines on your terminal to run the `deploy.js` script.

```
npx hardhat compile # Compiles the contracts
npx hardhat run scripts/deploy.js --network fuji # runs the "deploy.js" script on fuji test network, "fuji" is specified inside the hardhat config file
```

Pretty simple! If this looks unfamiliar to you, you may want to take a look at [hardhat guides](#).

Do not forget to note the addresses, as we will need them afterward to interact with the contracts. 

React App

Let's build an interface to interact with our marketplace. We are going to use `react` and `ether.js`.

Execute the following lines on your terminal to get started.

```
create-react-app frontend # creates a react app inside the frontend folder
cd frontend # go inside the frontend folder
npm install --save ethers # install ethers package
npm run start # start the react app
```

Add bootstrap CDN in the head section of `public/index.html` file. We will use bootstrap to speed up.

```
<head>
  ...
  + <link
  +   rel="stylesheet"
  +   href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
  + />
</head>
...
</body>
+ <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"></script>
+ <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"></script>
+ <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"></script>
</html>
```

Start with a fresh `App.js` file. Import the `ethers` library and assign the contract addresses to constant strings.

Since there aren't any auctions yet, we need to create one first. However, before creating an auction, we need to mint an NFT.

```
import React from "react";
import { ethers } from "ethers";
const NFT_ADDRESS = "0xeb2283672cf716fF6A1d880436D3a9074Ba94375"; // NFT contract address
const AUCTIONMANAGER_ADDRESS = "0xea4b168866E439Db4A5183Dcbc4951DCb5437f1E"; // AuctionManager contract address
class App extends React.Component {
  constructor(props) {
```

```

super(props);
this.state = {
  auctions: [], // Auctions to display
  newAuction: {
    // newAuction is a state variable for the form
    startPrice: null,
    endTime: null,
    tokenId: null,
    minIncrement: null,
    directBuyPrice: null,
  },
  myItems: [],
};
}
render() {
  return (
    <div>
      <div class="jumbotron d-flex align-items-center">
        <div class="container">
          <div class="auctions row"></div>
        </div>
      </div>
      <div class="container">
        <form>
          <button type="submit" class="btn btn-primary">
            Create Auction
          </button>
        </form>
        <button class="btn btn-fanger">Mint NFT</button>
        <p>
          Your items
          <br />
          {(this.state.myItems || []).map((x) => `id: ${x} `) || ""}
        </p>
      </div>
    </div>
  );
}
export default App;

```

Building the Form

We need to add few elements to our form. The user has to type in the start price, token id, minimum increment, how many minutes the auction will last and the direct buy price.

```

<form>
  <div class="mb-3">
    <label for="startprice" class="form-label">
      Start Price
    </label>
    <input
      value={this.state.newAuction.startPrice}
      onChange={(e) =>
        this.setState({
          newAuction: {
            ...this.state.newAuction,
            startPrice: parseInt(e.target.value),
          },
        })
      }
      type="number"
      class="form-control"
      id="startprice"
    />
    <label for="startprice" class="form-label">
      Token Id
    </label>
    <input
      value={this.state.newAuction.tokenId}
      onChange={(e) =>
        this.setState({
          newAuction: {
            ...this.state.newAuction,
            tokenId: parseInt(e.target.value),
          },
        })
      }
    />
  </div>

```

```

        }
        type="number"
        class="form-control"
        id="startprice"
    />
<label class="form-label">Minimum Increment</label>
<input
    value={this.state.newAuction.minIncrement}
    onChange={(e) =>
        this.setState({
            newAuction: {
                ...this.state.newAuction,
                minIncrement: parseInt(e.target.value),
            },
        })
    }
    type="number"
    class="form-control"
/>
<label class="form-label">Direct Buy Price</label>
<input
    value={this.state.newAuction.directBuyPrice}
    onChange={(e) =>
        this.setState({
            newAuction: {
                ...this.state.newAuction,
                directBuyPrice: parseInt(e.target.value),
            },
        })
    }
    type="number"
    class="form-control"
/>

<label class="form-label">Duration In Minutes</label>
<input
    value={this.state.newAuction.endTime}
    onChange={(e) =>
        this.setState({
            newAuction: {
                ...this.state.newAuction,
                endTime: parseInt(e.target.value),
            },
        })
    }
    type="number"
    class="form-control"
/>
</div>

<button
    type="button"
    onClick={() => console.log(this.state.newAuction)}
    class="btn btn-primary"
>
    Create Auction
</button>
</form>

```



Our form is ready! Let's see how we are going to interact with the contracts.

Interacting with Contracts

Contract ABIs

We will use Contract Application Binary Interface (ABI) to interact with the contracts. It is the standard way to do it. You can find the `ABI` of the compiled contracts in `artifacts/contracts/ContractName/ContractName.json` file. Create a `frontend/src/artifacts` folder and move all those `json` files there to reach them easier within the React code.

If you can not find the json files try running `npx hardhat compile` on your terminal

```

mkdir frontend/src/artifacts # creates a folder named "artifacts" inside the "src" folder of your React app
mv artifacts/contracts/Auction.sol/Auction.json frontend/src/artifacts/Auction.json # Moves Auction.json to the newly created folder.
mv artifacts/contracts/NFT.sol/NFT.json frontend/src/artifacts/NFT.json # Moves NFT.json to the newly created folder.

```

```
mv artifacts/contracts/AuctionManager.sol/AuctionManager.json frontend/src/artifacts/AuctionManager.json # Moves AuctionManager.json to the newly created folder.
```

Import them in the React code at the top of [App.js](#).

```
import AuctionArtifact from "./artifacts/Auction.json";
import AuctionManagerArtifact from "./artifacts/AuctionManager.json";
import NFTArtifact from "./artifacts/NFT.json";
```

Detecting and Connecting to MetaMask

Add an `init` function which we will call when the page is loaded.

```
class App extends React.Component {
  async init() {}
  ...
}
```

We will use [Ethers](#) to connect our MetaMask wallet and call the `init` function on `componentDidMount`.

```
class App extends React.Component {
  ...
  async init() {
    if (window.ethereum) {
      await window.ethereum.enable(); // Enable the Ethereum client
      this.provider = new ethers.providers.Web3Provider(window.ethereum); // A connection to the Ethereum network
      this.signer = this.provider.getSigner(); // Holds your private key and can sign things
      this.setState({ currentAddress: this.signer.getAddress() }); // Set the current address
      this._auctionManager = new ethers.Contract( // We will use this to interact with the AuctionManager
        AUCTIONMANAGER_ADDRESS,
        AuctionManagerArtifact.abi,
        this.signer
      );

      this._nft = new ethers.Contract( // We will use this to interact with the NFT contract
        NFT_ADDRESS,
        NFTArtifact.abi,
        this.signer
      );
    } else {
      alert("No wallet detected");
    }
  }
  componentDidMount() {
    this.init();
  }
  ...
}
```

Inside the `init` function we are creating instances of the signer, auction manager contract and our NFT contract. We will use them later.

[Take a look at here to learn more about Ethers](#). Refresh the page now, MetaMask should prompt you to connect your account.

 Choose account  Choose account next

Choose your account and continue. Congrats! We have connected MetaMask to our website.

Minting an NFT

Let's build the mint function, which we will call when the `Mint NFT` button is pressed.

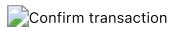
We are going to use the `this._nft` object which we created inside the `init` function earlier. There is a function named `getItem()` inside our NFT contract. We have to call it and pass 0.5 AVAX to mint an NFT.

```
async mint() {
  // hash is the hash of the transaction
  let { hash } = await this._nft.getItem({
    // Calling the getItem function of the contract
    value: ethers.utils.parseEther("0.5"), // 0.5 AVAX
  });
  console.log("Transaction sent! Hash:", hash);
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  console.log("Transaction mined!");
  alert(`Transaction sent! Hash: ${hash}`);
}
```

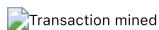
Call this function when the button is pressed.

```
<button type="button" onClick={() => this.mint()} class="btn btn-danger">
  Mint NFT
</button>
```

Then click the Mint NFT button on the web page.



Confirm the transaction. After a few seconds the transaction should get mined.



Congrats! We have just minted ourselves an NFT token! I will mint myself two more. You can do as many as you want!

Owned NFT's

We will use the `myItems` function in the NFT contract to retrieve a list of tokens that we own.

```
async.getItems() {
  let items = await this._nft.myItems(); // Get the tokens owned by the user
  console.log(items);
}
```

Call this function at the end of the `init()` function, so each time the wallet is connected, we will retrieve the list of items owned by that wallet.

```
async init() {
  if (window.ethereum) {
    // if window.ethereum is defined
    await window.ethereum.enable(); // Enable the Ethereum client
    this.provider = new ethers.providers.Web3Provider(window.ethereum); // A connection to the Ethereum network
    this.signer = this.provider.getSigner(); // Holds your private key and can sign things

    this._auctionManager = new ethers.Contract( // We will use this to interact with the AuctionManager
      AUCTIONMANAGER_ADDRESS,
      AuctionManagerArtifact.abi,
      this.signer
    );

    this._nft = new ethers.Contract( // We will use this to interact with the NFT
      NFT_ADDRESS,
      NFTArtifact.abi,
      this.signer
    );

    +   this.getItems();
  } else {
    alert("No wallet detected"); // No wallet detected
  }
}
```



If you reload the page and take a look at the developer console, you may notice that the array contains values in `BigNumber` format. We have to convert them to normal numbers.

```
async.getItems() {
  let items = await this._nft.myItems(); // Get the tokens owned by the user
  items = items.map((x) => x.toNumber()); // Converts BigNumber to number for each item in array
  this.setState({ myItems: items });
}
```

Now refresh the page.



Here they are! I can see all the NFT's I've minted.

Creating an Auction

So far, so good! Let's build a function for creating auctions. Before calling the `createAuction` function in the auction manager, first, we have to give approval for the token that we are going to auction. Giving approval to the auction manager will let the auction manager transfer our token to the newly created auction contract.

Here are the steps.

- Check if there are any empty text fields.

```
if (
  !this.state.newAuction.minIncrement ||
  !this.state.newAuction.directBuyPrice ||
  !this.state.newAuction.startPrice ||
  !this.state.newAuction.endTime ||
  !this.state.newAuction tokenId
)
  return alert("Fill all the fields");
```

- We call the `approve` function of the NFT contract to approve the `AuctionManager` contract to transfer the NFT token that we are going to put up for auction.

```
let { hash: allowance_hash } = await this._nft.approve(
  AUCTIONMANAGER_ADDRESS,
  this.state.newAuction tokenId
); // Approve the AUCTIONMANAGER to transfer the token
console.log("Approve Transaction sent! Hash:", allowance_hash);
await this.provider.waitForTransaction(allowance_hash); // Wait till the transaction is mined
console.log("Transaction mined!");
```

- Then, we will call the `createAuction` function in the `AuctionManager` contract.

```
let { hash } = await this._auctionManager.createAuction(
  this.state.newAuction.endTime * 60, // Converting minutes to seconds
  ethers.utils.parseEther(this.state.newAuction.minIncrement.toString()), // Minimum increment in AVAX
  ethers.utils.parseEther(this.state.newAuction.directBuyPrice.toString()), // Direct buy price in AVAX
  ethers.utils.parseEther(this.state.newAuction.startPrice.toString()), // Start price in AVAX
  NFT_ADDRESS, // The address of the NFT token
  this.state.newAuction tokenId // The id of the token
);
console.log("Transaction sent! Hash:", hash);
await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
console.log("Transaction mined!");
alert(`Transaction sent! Hash: ${hash}`);
```

Final function

```
async createAuction() {
  if (
    !this.state.newAuction.minIncrement ||
    !this.state.newAuction.directBuyPrice ||
    !this.state.newAuction.startPrice ||
    !this.state.newAuction.endTime ||
    !this.state.newAuction tokenId
  )
    return alert("Fill all the fields");

  let { hash: allowance_hash } = await this._nft.approve(
    AUCTIONMANAGER_ADDRESS,
    this.state.newAuction tokenId
  ); // Approve the AUCTIONMANAGER to transfer the token
  console.log("Approve Transaction sent! Hash:", allowance_hash);
  await this.provider.waitForTransaction(allowance_hash); // Wait till the transaction is mined
  console.log("Transaction mined!");

  let { hash } = await this._auctionManager.createAuction(
    // Create an auction
    this.state.newAuction.endTime * 60, // Converting minutes to seconds
    ethers.utils.parseEther(this.state.newAuction.minIncrement.toString()), // Minimum increment in AVAX
    ethers.utils.parseEther(this.state.newAuction.directBuyPrice.toString()), // Direct buy price in AVAX
    ethers.utils.parseEther(this.state.newAuction.startPrice.toString()), // Start price in AVAX
    NFT_ADDRESS, // The address of the NFT token
    this.state.newAuction tokenId // The id of the token
  );
  console.log("Transaction sent! Hash:", hash);
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  console.log("Transaction mined!");
```

```

    alert(`Transaction sent! Hash: ${hash}`);
}

```

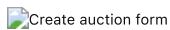
Call this function when the `Create Auction` button is pressed.

```

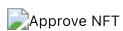
...
<button
  type="button"
+ onClick={() => this.createAuction()}
  class="btn btn-primary"
>
  Create Auction
</button>
...

```

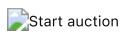
We are ready to start our first auction! Fill the form and press on `Create Auction` button.



Approve the auction manager contract to transfer the NFT token with ID 2.



Start the auction!



That's it, we have just started our first auction on our NFT marketplace.

Getting the List of Auctions

Let's build a function to get a list of all auctions.

We will use two functions from our `AuctionManager` contract to achieve this, `getAuctions()` and `getAuctionInfo()`.

By calling the `getAuctions()` function, we will get a list of contract addresses of all the auctions on the marketplace. Next, we will call the `getAuctionInfo()` function and pass an array of auction addresses that we would like to obtain information.

```

async getAuctions() {
  let auctionsAddresses = await this._auctionManager.getAuctions(); // get a list of auction addresses
  let auctions = await this._auctionManager.getAuctionInfo(auctionsAddresses); // I'll just pass all the addresses here, you can
  build a pagination system if you want
  console.log("Auctions:", auctions);
  this.setState({ auctions }); // Update the state
}

```

We should call this function at the end of the `init()` function.

```

...
this._nft = new ethers.Contract(NFT_ADDRESS, NFTArtifact.abi, this.signer); // We will use this to interact with the NFT
this.getItems();
+ this.getAuctions();
...

```

Now, take a look at the developer console logs.



The numbers are again in big number format. We can convert the `tokenId` and `endTime` using `.toNumber()` method; however, for the `BigNumber`s that represent a price, we should use `ethers.utils.formatEther` to get the exact value in AVAX.

We should organize the auction array and display them in the `auctions` section of the web page.

```

let new_auctions = [];

for (let i = 0; i < auctions.endTime.length; i++) {
  let endTime = auctions.endTime[i].toNumber();
  let tokenId = auctions.tokenIds[i].toNumber();
  let auctionState = auctions.auctionState[i].toNumber();

  let startPrice = ethers.utils.formatEther(auctions.startPrice[i]);
  let directBuyPrice = ethers.utils.formatEther(auctions.directBuy[i]);
  let highestBid = ethers.utils.formatEther(auctions.highestBid[i]);
}

```

```

let owner = auctions.owner[i];

let newAuction = {
  endTime: endTime,
  startPrice: startPrice,
  owner: owner,
  directBuyPrice: directBuyPrice,
  tokenId: tokenId,
  highestBid: highestBid,
  auctionState: auctionState,
  auctionAddress: auctionsAddresses[i],
};

new_auctions.push(newAuction);
}

this.setState({ auctions: new_auctions }); // Update the state
}

```

Now we will write a function to display an auction on the UI.

```

renderAuctionElement(auction) {
  let state = "";
  if (auction.auctionState === 0) {
    state = "Open";
  }
  if (auction.auctionState === 1) {
    state = "Cancelled";
  }
  if (auction.auctionState === 2) {
    state = "Ended";
  }
  if (auction.auctionState === 3) {
    state = "Direct Buy";
  }
  return (
    <div style={{ background: "yellow" }} class="col">
      <p>ID: {auction.tokenId}</p> /* ID of the token */
      <p>Highest Bid: {auction.highestBid || 0}</p>
      /* Highest bid */
      <p>Direct Buy: {auction.directBuyPrice}</p> /* Direct buy price */
      <p>Starting Price: {auction.startPrice}</p> /* Starting price */
      <p>Owner: {auction.owner}</p> /* Owner of the token */
      <p>
        End Time: {(Math.round((auction.endTime * 1000 - Date.now()) / 1000 / 60))}(" ")
        /* Time left in minutes */
        minutes
      </p>
      <p>Auction State: {state}</p>
      <button class="btn-primary">See More</button>
    </div>
  );
}

```

We have converted the `auction.endTime`, which is in seconds, to milliseconds by multiplying by 1000. Then, we have subtracted the result from `Date.now()`, which is in milliseconds. After that, we divided the result by 1000 to convert it to seconds. Finally, we divided it by 60 to convert it to minutes.

Next, we will map all the auctions into the function in our `render` method.

```

render() {
  return (
    <div class="jumbotron d-flex align-items-center">
      <div class="container">
        <div class="auctions row">
          {this.state.auctions.map(this.renderAuctionElement)}
        </div>
      </div>
    </div>
  )
}

```

Let's see the result!



Awesome! There is the auction we've just created.

Building the Auction Page

We'll use a state variable `this.state.activeAuction` to display the current auction. We will set this variable when the `See More` button is pressed. If this variable is not null, the auction information will be rendered instead of the list of auctions. We will set this variable back to null by using a go-back button.

This is not a good practice to do it this way, you should probably use [React Navigation](#), but since this is just a demo, we are doing it this way.

Add a new `setActiveAuction` function

```
setActiveAuction(auction) {
  this.setState({ activeAuction: auction });
}
```

It will call the function when the button is pressed.

```
...
<p>Auction State: {state}</p>
<button class="btn-primary" onClick={() => this.setActiveAuction(auction)}>See More</button>
</div>
...
```

We would like to display the previous bids for a specific auction. Let's fetch that data when the `setActiveAuction` is called.

```
async setActiveAuction(auction) {
  this._auction = new ethers.Contract( // Create a new instance of the contract
    auction.auctionAddress,
    AuctionArtifact.abi,
    this.signer
  );

  let previousBids = await this._auction.allBids(); // Get the bids
  let bids = []; // A list of bids
  for (let i = 0; i < previousBids[0].length; i++) { // Loop through the bids
    bids.push({ // Add the bid to the list
      bidder: previousBids[0][i], // The bidder
      bid: ethers.utils.formatEther(previousBids[1][i]), // The bid
    });
  }

  auction.bids = bids; // Add the bids array to the auction object

  this.setState({ activeAuction: auction }); // Update the state
}
```

Later, we will use the `this._auction` to interact with the auction contract. That's why we have made it accessible from `this`.

Do you remember that we have assigned a random value to every minted NFT? Let's also display that value on the auction page.

```
let auctionTokenValue = await this._nft.itemValue(auction tokenId); // Get the value of the token
auctionTokenValue = auctionTokenValue.toNumber(); // Convert BigNumber to number
auction.auctionTokenValue = auctionTokenValue; // Add the value of the token to the auction object
```

We should also know who is the highest bidder.

```
let highestBidder = await this._auction.maxBidder(); // Get the highest bidder
auction.highestBidder = highestBidder; // Add the highest bidder to the auction object
```

Lastly, the minimum increment value.

```
let minIncrement = await this._auction.minIncrement(); // Get the minimum increment
auction.minIncrement = ethers.utils.formatEther(minIncrement); // Add the minimum increment to the auction object
```

Here is the complete function.

```
async setActiveAuction(auction) {
  console.log(auction);
  this._auction = new ethers.Contract( // Create a new instance of the contract
    auction.auctionAddress,
    AuctionArtifact.abi,
    this.signer
  );

  let previousBids = await this._auction.allBids(); // Get the bids
  let bids = []; // A list of bids
```

```

for (let i = 0; i < previousBids[0].length; i++) {
  // Loop through the bids
  bids.push({
    // Add the bid to the list
    bidder: previousBids[0][i], // The bidder
    bid: ethers.utils.formatEther(previousBids[1][i]), // The bid
  });
}

auction.bids = bids; // Add the bids array to the auction object

let auctionTokenValue = await this._nft.itemValue(auction.tokenId); // Get the value of the token
auctionTokenValue = auctionTokenValue.toNumber(); // Convert BigNumber to number
auction.auctionTokenValue = auctionTokenValue; // Add the value of the token to the auction object

let highestBidder = await this._auction.maxBidder(); // Get the highest bidder
auction.highestBidder = highestBidder; // Add the highest bidder to the auction object

let minIncrement = await this._auction.minIncrement(); // Get the minimum increment
auction.minIncrement = ethers.utils.formatEther(minIncrement); // Add the minimum increment to the auction object

this.setState({ activeAuction: auction }); // Update the state
}

```

Rendering the Selected Auction

We will write a simple render function for the selected auction.

```

renderActiveAuction() {
  let activeAuction = this.state.activeAuction;
  let state = "";
  if (activeAuction.auctionState === 0) { // If the auction is open
    state = "Open";
  }
  if (activeAuction.auctionState === 1) { // If the auction is cancelled
    state = "Cancelled";
  }
  if (activeAuction.auctionState === 2) { // If the auction is ended
    state = "Ended";
  }
  if (activeAuction.auctionState === 3) { // If the auction is ended by a direct buy
    state = "Direct Buy";
  }
  return (
    <div>
      <div class="col">
        <button
          class="btn-secondary"
          onClick={() => this.setState({ activeAuction: null })}
        >
          Go Back
        </button>
        <p>ID: {activeAuction.tokenId}</p> /* ID of the token */
        <p>Highest Bid: {activeAuction.highestBid || 0} AVAX</p>
        /* Highest bid */
        <p>Direct Buy: {activeAuction.directBuyPrice} AVAX</p> ""
        /* Direct buy price */
        <p>Minimum Increment: {activeAuction.minIncrement} AVAX</p> ""
        /* Minimum increment in AVAX */
        <p>Starting Price: {activeAuction.startPrice} AVAX</p> /* Starting price */
        <p>Owner: {activeAuction.owner}</p> /* Owner of the token */
        <p>
          End Time:(" ")
          (Math.round(
            (activeAuction.endTime * 1000 - Date.now()) / 1000 / 60
          ))(" ")
        /* Time left in minutes */
        minutes
        </p>
        <p>Auction State: {state}</p>
      </div>
      <div class="col">
        <h3>Bids</h3>
        <table class="table">
          <thead>
            <tr>
              <th>Bidder</th>

```

```

        <th>Bid</th>
    </tr>
</thead>
<tbody>
    {activeAuction.bids.map((bid) => {
        return (
            <tr key={bid.bidder}>
                <td>{bid.bidder}</td>
                <td>{bid.bid} AVAX</td>
            </tr>
        );
    ))}
</tbody>
</table>
</div>
<div class="col">
    <div>
        <input type="number" placeholder="0.5" />
        <button class="btn-primary">Place Bid</button>
    </div>
    <button class="btn-danger">Cancel Auction</button>
    <button class="btn-secondary">Withdraw Funds</button>
    <button class="btn-secondary">Withdraw Token</button>
    </div>
</div>
);
}

```

If `this.state.activeAuction` is not null, then we will return this function instead of the list of auctions.

Edit the `render` function

```

...
<div class="container">
    {this.state.activeAuction != null ? (
        this.renderActiveAuction()
    ) : (
        <div class="auctions row">
            {this.state.auctions.map(this.renderAuctionElement)}
        </div>
    )
</div>
...

```



Now, we will have to make those buttons at the bottom functional. Let's write a function for each one!

Auction Functions

Remember the `this._auction` object we've set inside the `setActiveAuction` function? We will use that object to build our functions.

Placing Bids

We will simply call the `placeBid` function of the Auction contract and pass some AVAX.

```

async placeBid(amount) {
    if (!amount) return;
    amount = ethers.utils.parseEther(amount.toString()); // Amount in AVAX
    let { hash } = await this._auction.placeBid({ value: amount }); // Place a bid
    await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
    alert(`Transaction sent! Hash: ${hash}`); // Show the transaction hash
    this.setActiveAuction(this.state.activeAuction); // Update the active auction
}

```

Next, call this function when the button is pressed.

```

...
<div>
    <input
        type="number"
        placeholder="0.5"
        onChange={(e) => this.setState({ bidAmount: e.target.value })}
    />
    <button
        class="btn-primary"

```

```

    onClick={() => this.placeBid(this.state.bidAmount)}
>
  Place Bid
</button>
</div>
...

```

Before placing a bid, we must switch to another account on MetaMask since the auction creator cannot place bids. Do not forget to connect the account to the 1537dapp when you switch your account.



We've placed a bid on the auction! Now let's wait a bit until the auction is over, then we can withdraw our new token if no one bids a higher value.

Withdraw Token

Simply call the `withdrawToken` function of the Auction contract and show an alert when the transaction gets mined.

```

async withdrawToken() {
  let { hash } = await this._auction.withdrawToken(); // Withdraw the NFT token
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  alert('Withdrawal Successful! Hash: ${hash}'); // Show the transaction hash
  window.location.reload(); // Reload the page
}

```

Call this function when the `Withdraw Token` button is pressed.

```

...
<button onClick={()=>this.withdrawToken()} class="btn-secondary">Withdraw Token</button>
...

```



After some time, the auction is finally over, and our bid is still the highest one. Let's withdraw our new NFT token. Note: It would be nice to hide the end time when the auction is not open.

Approving the transaction...



Refresh the page and we just got our new NFT!



Withdraw Funds

Simply call the `withdrawFunds` function of the Auction contract and show an alert when the transaction gets mined.

```

async withdrawFunds() {
  let { hash } = await this._auction.withdrawFunds(); // Withdraw the funds
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  alert('Withdrawal Successful! Hash: ${hash}'); // Show the transaction hash
  window.location.reload(); // Reload the page
}

```

Call this function when the `Withdraw Funds` button is pressed.

```

...
<button onClick={()=>this.withdrawFunds()} class="btn-secondary">Withdraw Funds</button>
...

```

Since the auction is already over, we can switch back to the account in which we created the auction and withdraw our funds.



Then we will click on the `Withdraw Funds` button and confirm the transaction.



Awesome! We have just sold an NFT and earned some AVAX.

Cancel Auction

Simply call the `cancelAuction` function of the Auction contract and show an alert when the transaction gets mined.

```
async cancelAuction() {
  let { hash } = await this._auction.cancelAuction(); // Cancel the auction
  await this.provider.waitForTransaction(hash); // Wait till the transaction is mined
  alert(`Auction Canceled! Hash: ${hash}`); // Show the transaction hash
  window.location.reload(); // Reload the page
}
```

Call this function when the `Cancel Auction` button is pressed.

```
...
<button onClick={() => this.cancelAuction()} class="btn-danger">
  Cancel Auction
</button>
...
```

We have to create a new auction to demonstrate this since we cannot cancel an auction that has been bid already. Create a new auction, click on the `Cancel Auction` button and confirm the transaction.

After refreshing the page, we can see that the auction has been canceled.



We can also see that the token was sent back to use after we've canceled the auction.

Polishing the Buttons

If the button is not functional we should hide it. For instance, the `Withdraw Funds` button is visible to all users; however, only the auction creator can call it when the auction is over. Thus, we must only make it visible if the auction is over and the user is the auction creator.

We will do some checks for each button and hide the button if the user is not able to call the function.

Please note that our contract already blocks the other users from calling `Withdraw Funds` function. Someone can call any external function of a contract, even if it's hidden on the web page. It is crucial to make the checks within the contract code.

We will use some booleans to make this easier.

```
renderActiveAuction() {
  let activeAuction = this.state.activeAuction;

  let isOwner = this.state.currentAddress === activeAuction.owner; // Check if the current address is the owner
  let isAuctionOpen = state === "Open"; // Check if the auction is open
  let isAuctionCancelled = state === "Cancelled"; // Check if the auction is cancelled
  let isAuctionEnded = state === "Ended" || state === "Direct Buy"; // Check if the auction is ended
  ...
}
```

Place Bid Area

- Auction must be open.
- The auction creator cannot bid.

```
{
  isAuctionOpen && !isOwner ? (
    <div>
      <input
        type="number"
        placeholder="0.5"
        onChange={(e) => this.setState({ bidAmount: e.target.value })}
      />
      <button
        class="btn-primary"
        onClick={() => this.placeBid(this.state.bidAmount)}
      >
        Place Bid
      </button>
    </div>
  ) : null;
}
```

Cancel Auction Button

- There must be no bids.

- Auction must be open.
- The caller must be the auction creator.

```
{
  isOwner && isAuctionOpen && activeAuction.bids.length == 0 ? (
    <button onClick={() => this.cancelAuction()} class="btn-danger">
      Cancel Auction
    </button>
  ) : null;
}
```

Withdraw Funds Button

- Auction must be ended.
- The caller must be the auction creator.
- There must be at least one bid on the auction.

```
{
  isOwner && isAuctionEnded && activeAuction.bids.length > 0 ? (
    <button onClick={() => this.withdrawFunds()} class="btn-secondary">
      Withdraw Funds
    </button>
  ) : null;
}
```

Withdraw Token Button

- Auction must be ended.
- If there are any bids, the caller must be the highest bidder.
- If there are no bids, the caller must be the auction creator. (The auction creator must be able to withdraw the token if no one bids and the auction times out)

```
{
  ((activeAuction.bids.length == 0 && isOwner) || isHighestBidder) &&
  isAuctionEnded ? (
    <button onClick={() => this.withdrawToken()} class="btn-secondary">
      Withdraw Token
    </button>
  ) : null;
}
```

See the code for all of the buttons

```
<div class="col">
  {isAuctionOpen && !isOwner ? (
    <div>
      <input
        type="number"
        placeholder="0.5"
        onChange={(e) => this.setState({ bidAmount: e.target.value })}
      />
      <button
        class="btn-primary"
        onClick={() => this.placeBid(this.state.bidAmount)}
      >
        Place Bid
      </button>
    </div>
  ) : null}
  {isOwner && isAuctionOpen && activeAuction.bids.length == 0 ? (
    <button onClick={() => this.cancelAuction()} class="btn-danger">
      Cancel Auction
    </button>
  ) : null}

  {isOwner && isAuctionEnded ? (
    <button onClick={() => this.withdrawFunds()} class="btn-secondary">
      Withdraw Funds
    </button>
  ) : null}
  {isHighestBidder && isAuctionEnded ? (
    <button onClick={() => this.withdrawToken()} class="btn-secondary">
      Withdraw Token
    </button>
  ) : null}
</div>
```

```
) : null}  
</div>
```

Finishing the React App

That's all we need to do. Now, we have a functional marketplace where we can create auctions and sell NFT's. We had an attribute called `value` that is assigned randomly to each of the NFT's. We've displayed this attribute on the auction page.

If this were an image or any attribute else instead of an integer, we would still be able to display it by making a simple change in our code. For instance, to retrieve the metadata (image URL and other attributes) of an NFT token, use the `tokenURI(uint256 tokenId)` function to get the token URI from the NFT contract. You may want to take a look at [IERC721Metadata](#) to learn more about it.

Using with Other NFT Contracts

Selling another NFT instead of creating a contract is possible. As you may have noticed, we have only used the `approve` and `transferFrom` functions to transfer the NFT to our Market Place contract. Those functions are a standard in ERC-721, so this marketplace is compatible with almost any NFT token.

Next Steps

There are a couple of things to test and consider before going any further.

Tests

- Does the direct buy working?
- Will the auction creator be able to withdraw the token if there are no bids and the auction times out?
- Does the minimum increment working correctly?

Before deploying a contract to production, we should test every possible scenario, and it's very hard to do the test one by one on the UI. Instead, we should use [hardhat tests](#), where we can create different scenarios and test all of them in a matter of seconds.

Things to Consider

- If the user sends more AVAX than the direct buy price, it doesn't get refunded.
 - Refund the extra AVAX if `msg.value` is greater than direct buy price.
- The website won't work if the user doesn't connect a wallet or doesn't have one.
 - To be able to call the view functions, such as getting a list of auctions, you can use a `provider`, [learn more about it here](#)
- React App doesn't handle the error when a transaction fails.
 - When the transaction fails, maybe show a pop-up and tell the reason.

It's always good to give the user clear instructions and make them feel comfortable. You should look from a user's point of view and try to think of every possible scenario & outcome.

Deploying to the Avalanche Mainnet

Deploying to Mainnet is the same as deploying to [Testnet](#). The only difference is that you have to pay real funds instead of test funds.

Again, get the configurations for the Avalanche Mainnet from [here](#) and add the network in the hardhat config file [hardhat.config.js](#).

```
networks:{  
  ...  
  mainnet: {  
    url: "https://api.avax.network/ext/bc/C/rpc",  
    chainId: 43114,  
    accounts: [  
      "PRIVATE_KEY",  
    ],  
  },  
}
```

After that, run the deploy script just like when deploying to the test net.

```
npx hardhat compile # Compiles the contracts  
npx hardhat run scripts/deploy.js --network mainnet # runs the script on the Avalanche Mainnet, "mainnet" is specified inside  
the hardhat config file
```

Conclusion

You now have the basic knowledge to start your NFT Marketplace.

Don't forget that the react app is a demonstration of interacting with the contracts and fetching data from them. A good marketplace would need a better design and a lot of work.

Open an issue if you have any questions.

How to Create Your Own DAO with Avalanche

What Is a DAO

The DAO's are systems that help us to work with people around the world in a safe and clear way.

Think of them like an internet-native business that's collectively owned and managed by its members. They have built-in treasuries that no one has the authority to access without the approval of the group. Decisions are governed by proposals and voting to ensure everyone in the organization has a voice.

There's no CEO who can authorize spending based on their own whims and no chance of a dodgy CFO manipulating the books. Everything is out in the open and the rules around spending are baked into the DAO via its code.

How the DAO's Work?

The backbone of a DAO is in smart contracts. The smart contracts defines the rules of the organization.

Generally the DAO consists of two main smart contracts: the ERC20 token that is a governance token, and the smart contract that have the rules for the DAO. So like a DAO's members you need to have some governance tokens and then deposit to the DAO contract, and then we can create a proposal if the proposal is accepted the other members in the DAO can start to vote.

The vote is based in how many governance tokens have been deposited on the DAO.

For example if you have a 100 of the governance tokens but you deposited only 20 tokens on the DAO contract only 20 tokens will be taken into account for your vote.

Let's Start to Build Our DAO

- Tools We will use
 - [REMIX IDE](#)
 - [MetaMask Wallet](#)

We need to setup the Fuji Testnet on our MetaMask. [find here the rpc values](#)

Step 1: Creating a New .sol File on REMIX

On REMIX we click the new file icon and put some name, in my case my file name is `MyDAO.sol`



and we add the basic lines of code:

The first line tells you that the source code is licensed under the GPL version 3.0. Machine-readable license specifiers are important in a setting where publishing the source code is the default.

```
pragma Specifies that the source code is written for Solidity version 0.7.0 or a newer version of the language up to, but not including version 0.9.0.  
contract MyDAO {...} specifies the name and a new block of code for our contract.
```



Step 2: Defining Our DAO Functions

Commonly the DAO's contract has four main functions:

- Deposit governance tokens.
- Withdraw the tokens.
- Create a proposal.
- Vote.

We use AVAX our governance token. Fuji contract address: 0xA048B6a5c1be4b81d99C3Fd993c98783adC2eF70 and we need import [IERC20](#) template from [OpenZeppelin](#).

Step 3: Defining the Proposal Variables

For the proposal format we defined a group with custom properties, the properties for our proposal are:

- Author which is an address from the account that create a proposal.
- Id that will help us to identify a proposal.
- Name of the proposal.
- Creation date, that allow us to set a period of time for allow the voting.
- Voting options, in this case we will keep it simple(Yes / NO).
- Number of Votes for Yes and Votes for No this will allow us set an status for the proposal when number of votes for any option be greater than fifty percent.
- Status for the Proposal this options will be Accepted, Rejected, Pending.

For the voting options and the proposal status we will use an `enums` types.

`Enums` can be used to create custom types with a finite set of 'constant values'. [see more about Enums](#)

```
enum VotingOptions { Yes, No }
enum Status { Accepted, Rejected, Pending }
```

For the other proposal properties we can use an `struct` type.

Structs allow us to define a custom group of properties. [see more about structs](#)

```
struct Proposal {
    uint256 id;
    address author;
    string name;
    uint256 createdAt;
    uint256 votesForYes;
    uint256 votesForNo;
    Status status;
}
```

Until this step our DAO contract looks like this:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol';

contract MyDAO {

    enum VotingOptions { Yes, No }
    enum Status { Accepted, Rejected, Pending }
    struct Proposal {
        uint256 id;
        address author;
        string name;
        uint256 createdAt;
        uint256 votesForYes;
        uint256 votesForNo;
        Status status;
    }
}
```

Now we need to store all the proposals created for our DAO, we need to be sure that someone does not vote more than once, also set a period of vote for the proposals and set a minimum number of governance tokens to create a new proposal, we can take the number of governance tokens are deposited like a shares for an shareholder and give a proportional weight to their vote.

```
// store all proposals
mapping(uint => Proposal) public proposals;
// who already votes for who and to avoid vote twice
mapping(address => mapping(uint => bool)) public votes;
// one share for governance tokens
mapping(address => uint256) public shares;
uint public totalShares;
// the IERC20 allow us to use avax like our governance token.
IERC20 public token;
// the user need minimum 25 AVAX to create a proposal.
uint constant CREATE_PROPOSAL_MIN_SHARE = 25 * 10 ** 18;
uint constant VOTING_PERIOD = 7 days;
uint public nextProposalId;
```

Step 4: Deposit and Withdraw function for the DAO

We already have our necessary variables to create, save and vote a proposal in our DAO, now we need our user deposit his `AVAX` tokens to avoid that the same user can use the same amount of tokens for vote other option in the same proposal. To interact with AVAX as our token the governance we need to initialize the token address in the constructor.

```
constructor() {
    token = IERC20(0xA048B6a5c1be4b81d99C3Fd993c98783adC2eF70); // AVAX address
}
```

For the deposit function.

```
function deposit(uint _amount) external {
    shares[msg.sender] += _amount;
    totalShares += _amount;
```

```

        token.transferFrom(msg.sender, address(this), _amount);
    }
}

```

And we need to allow our users to withdraw their tokens when the voting period is over.

```

function withdraw(uint _amount) external {
    require(shares[msg.sender] >= _amount, 'Not enough shares');
    shares[msg.sender] -= _amount;
    totalShares -= _amount;
    token.transfer(msg.sender, _amount);
}

```

until this point our smart contract look like this:

```

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import 'https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol';

contract MyDAO {

    enum VotingOptions { Yes, No }
    enum Status { Accepted, Rejected, Pending }
    struct Proposal {
        uint256 id;
        address author;
        string name;
        uint256 createdAt;
        uint256 votesForYes;
        uint256 votesForNo;
        Status status;
    }

    // store all proposals
    mapping(uint => Proposal) public proposals;
    // who already votes for who and to avoid vote twice
    mapping(address => mapping(uint => bool)) public votes;
    // one share for governance tokens
    mapping(address => uint256) public shares;
    uint public totalShares;
    // the IERC20 allow us to use avax like our governance token.
    IERC20 public token;
    // the user need minimum 25 AVAX to create a proposal.
    uint constant CREATE_PROPOSAL_MIN_SHARE = 25 * 10 ** 18;
    uint constant VOTING_PERIOD = 7 days;
    uint public nextProposalId;

    constructor() {
        token = IERC20(0xA048B6a5c1be4b81d99C3Fd993c98783adC2eF70); // AVAX address
    }

    function deposit(uint _amount) external {
        shares[msg.sender] += _amount;
        totalShares += _amount;
        token.transferFrom(msg.sender, address(this), _amount);
    }

    function withdraw(uint _amount) external {
        require(shares[msg.sender] >= _amount, 'Not enough shares');
        shares[msg.sender] -= _amount;
        totalShares -= _amount;
        token.transfer(msg.sender, _amount);
    }
}

```

Step 5: Create a Proposal and Vote functions

For our `createProposal` function we will add the condition that if the user does not have minimum 25 AVAX tokens He cannot create a new proposal.

```

function createProposal(string memory name) external {
    // validate the user has enough shares to create a proposal
    require(shares[msg.sender] >= CREATE_PROPOSAL_MIN_SHARE, 'Not enough shares to create a proposal');

    proposals[nextProposalId] = Proposal(
        nextProposalId,

```

```

        msg.sender,
        name,
        block.timestamp,
        0,
        0,
        Status.Pending
    );
    nextProposalId++;
}

```

For the `Vote` function we need to receive the id for the proposal and the vote choice, we will validate that the user has not voted already and the vote period is currently open.

Also we validate if the proposal has more than fifty percent of votes in one option we need to change the proposal status to Accepted or Rejected.

```

function vote(uint _proposalId, VotingOptions _vote) external {
    Proposal storage proposal = proposals[_proposalId];
    require(votes[msg.sender][_proposalId] == false, 'already voted');
    require(block.timestamp <= proposal.createdAt + VOTING_PERIOD, 'Voting period is over');
    votes[msg.sender][_proposalId] = true;
    if(_vote == VotingOptions.Yes) {
        proposal.votesForYes += shares[msg.sender];
        if(proposal.votesForYes * 100 / totalShares > 50) {
            proposal.status = Status.Accepted;
        }
    } else {
        proposal.votesForNo += shares[msg.sender];
        if(proposal.votesForNo * 100 / totalShares > 50) {
            proposal.status = Status.Rejected;
        }
    }
}

```

Finally our DAO contract looks like this.



Step 6: Deploy Our DAO contract on Fuji

Now we need compile our contract, I'm using the 0.8.0 version compiler, and click on the `Compile` button.



In the environment section we choose the `Injected Web3` option, in account we chose an account from our MetaMask plugin in the Fuji network, make sure that your account have the necessary AVAX for the deploy and the minimum for create a proposal. [Here you can find the Faucet](#).

Click on the `Deploy` button and confirm the transaction in REMIX and MetaMask and await for a few seconds.



If the contract is deployed successfully on Fuji we can see the success transaction on the REMIX inspector.



Now we can test the different functions for our DAO.

Avalanche Developer ERC721 Tutorial

Introduction

In this tutorial you will get familiar with [ERC721\(NFT\)](#) smart contracts and how to deploy these to the [Avalanche Fuji testnet](#) and also the [Avalanche Mainnet \(C-Chain\)](#). The goal of this tutorial is to be as beginner friendly as possible. I will go through each line of code in order to give you a full understanding of what is happening, so that you can use the concepts as a basis for your first NFT decentralized application. The plan is to showcase:

1. How to create an ERC721 smart contract, so that you can mint your own ERC721 NFT on Avalanche using [Open Zeppelin](#) and the [Truffle framework](#);
2. How to extend the contract, so that each token has royalties;
3. How to create your own NFT marketplace where you can list your items, cancel listings and purchase other NFTs;
4. How to extensively test your smart contracts using Truffle's built in Mocha.js library and Open Zeppelin's Test Helper assertion library achieving 100% code coverage;

5. How to deploy your smart contracts on the Avalanche Fuji testnet and on the Avalanche Mainnet;

Development

For this tutorial I have used [Visual Studio Code](#) as my code editor of choice but you can use any code editor you like. I recommend to install the **solidity** extension by Juan Blanco to get that nice Syntax highlighting along with some code snippets should you go for Visual Studio Code. You would also need to create a MetaMask wallet or whatever similar provider you are comfortable with in order to deploy to the networks.

Dependencies

- [NodeJS v8.9.4 or later](#).
- Truffle, which you can install with npm install -g truffle
- (Optional) [Avalanche Network Runner](#) is a tool for running a local Avalanche network.

Setting Up a Truffle Project

1. Create a project directory and inside of it run the commands:

```
truffle init -y  
npm init -y
```

2. The first one will provide you with a base structure of a Truffle project and the second one will include a **package.json** file to keep track of the dependencies. Afterwards, include the following dependencies which will help us build and test the smart contracts.

```
npm install @openzeppelin/contracts @truffle/hdwallet-provider dotenv typescript typechain truffle-typings ts-node  
npm install --save-dev @openzeppelin/test-helpers solidity-coverage
```

- The [@openzeppelin/contracts](#) is a library for a secure smart contract development. We inherit from their ERC721 smart contract;
- The [@truffle/hdwallet-provider](#) is used to sign transactions for addresses derived from a 12 or 24 word mnemonic. In our case we will create a MetaMask wallet and provide the mnemonic from there to deploy to the Avalanche Fuji testnet;
- The [dotenv](#) is a zero-dependency module that loads environment variables from a .env file into process.env. We do not want to leak our mnemonic to other people after all;
- The [typechain](#) allows us to use TypeScript within Truffle;
- The [truffle-typings](#) is a library that goes with TypeChain should you want to use TypeScript for your Truffle environment;
- The [ts-node](#) is a package which we would need for a TypeScript execution of our scripts in Node.js;
- The [@openzeppelin/test-helpers](#) is a library that will help us test when transactions revert and also handle Big Numbers for us. It is a dev dependency;
- The [solidity-coverage](#) is a library that we will use to check how much coverage our tests have. It is again a dev dependency;

3. Now that we have all the necessary dependencies installed, let us go to the **truffle-config.js** in the root of our project and paste the following lines of code in there:

```
//we need this to be able to run our .ts tests  
require('ts-node/register')  
  
const HDWalletProvider = require('@truffle/hdwallet-provider')  
require('dotenv').config()  
module.exports = {  
  networks: {  
    fuji: {  
      provider: () => {  
        return new HDWalletProvider({  
          mnemonic: process.env.MNEMONIC,  
          providerOrUrl: `https://avalanche--fuji--rpc.datahub.figment.io/apikey/${process.env.APIKEY}/ext/bc/C/rpc`,  
          chainId: '43113'  
        })  
      },  
      network_id: "*",  
      gasPrice: 225000000000  
    },  
    mainnet: {  
      provider: () => {  
        return new HDWalletProvider({  
          mnemonic: process.env.MNEMONIC,  
          providerOrUrl: `https://api.avax.network/ext/bc/C/rpc`,  
          chainId: '43114',  
        })  
      },  
      network_id: "*",  
      gasPrice: 225000000000  
    }  
  },  
  compilers: {  
    solc: {  
      version: "0.8.6"  
    }  
  },  
  plugins: ["solidity-coverage"],
```

```

    test_file_extension_regex: /\.ts$/
}

```

This file is the entrypoint of our Truffle project. As you can see, we specify two networks on which we would like to deploy our smart contracts after we are done with them, namely *Fuji* and *Mainnet*. We utilize the [@truffle/hdwallet-provider](#) library, so that we provide a RPC to which we can connect to in order to deploy our contracts as well as a mnemonic which will be used to sign the transaction to do that. As you can see, some of the variables are accessed via process.env. These are defined in a separate **.env** file in the root of our project and have the following structure:

```

MNEOMNIC='paste your metamask mnemonic here which is twelve words long believe me'
APIKEY=YOUR_DATAHUB_API_KEY_FOR_THE_FUJI_TESTNET

```

Note: For the Fuji testnet I used [DataHub](#)'s testnet RPC. There is a free plan which you can use. For that you would need to register, grab your APIKEY and paste it into your **.env** file. Complete DataHub onboarding guide to DataHub 2.0 can be found [HERE](#)

4. We would also need two more configuration files in order to build our initial TypeScript environment. Let us create a **tsconfig.json** under the root of our project with the following contents:

```

{
  "compilerOptions": {
    "target": "es2017",
    "module": "commonjs",
    "strict": true,
    "moduleResolution": "node",
    "noImplicitThis": true,
    "noImplicitAny": false,
    "alwaysStrict": true,
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "forceConsistentCasingInFileNames": true,
    "lib": [
      "es2015"
    ],
    "sourceMap": true,
    "types": [
      "node",
      "truffle-typings"
    ]
  },
  "include": [
    "**/*.ts"
  ],
  "exclude": [
    "node_modules",
    "build"
  ]
}

```

As well as a **tsconfig.migrate.json** again under the root folder of our project:

```

{
  "extends": "./tsconfig.json",
  "include": [
    "./migrations/*.ts"
  ]
}

```

This file simply extends our base **tsconfig.json** by taking the folder **migrations/** contents into account. We will go through migrating in more detail after we have finished implementing and testing our contracts.

5. Add `generate` script in your **package.json**:

```

"scripts": {
  "generate": "typechain --target=truffle-v5 'build/contracts/*.json'"
}

```

Whenever you make changes to your smart contract you would need to run `npm run generate`.

Writing Our First ERC721 Contract

- Inside the **contracts/** folder of your Truffle project we will create a new [Collectible.sol](#) file and implement the functions we need. Now that we have this, let us start from the top and explain what the smart contract does.
- [Collectible.sol](#) logic
 - At the top we define the [Solidity version 0.8.6](#) which at the time of this tutorial is the latest one:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.6;
```

- Next we import the [ERC721URIStorage.sol](#) contract from Open Zeppelin. This contract is an extension of their [ERC721.sol](#) contract which takes metadata of an NFT into account as well. We will also use the popular [SafeMath.sol](#) library for our mathematical operations. This is a library which prevents unsigned integer overflows:

```
import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
```

- Now that we have done that let us define our **Collectible** contract which will inherit from the **ERC721URIStorage**. This would allow us to use all the functions and access all the public state variables which that contract offers. Pretty cool, right?

```
contract Collectible is ERC721URIStorage {
```

- Afterwards we will define the state variables and also the constructor of our contract:

```
mapping(string => bool) public hasBeenMinted;
mapping(uint256 => Item) public tokenIdToItem;
struct Item {
    address owner;
    address creator;
    uint256 royalty;
}
Item[] private items;
event ItemMinted(uint256 tokenId, address creator, string metadata, uint256 royalty);

constructor() ERC721("NFTCollectible", "NFTC") {}
```

The main thing to note here is the **Item** struct. We need one to keep track of some extra on-chain data that our NFTs can have. In our case this is the **owner**, the **creator** and the **royalty** which would be a percentage of the price paid out to the **creator** on each purchase of the NFT. With each minting, a new **Item** will be pushed to the array of items. This array can be then used to display those properties on the frontend, for example. We define an **ItemMinted** event due to our custom NFTs which we will emit at the end of our minting function. Last but not least, as you can see, we initialize the ERC721 constructor by providing it a name for our token contract and a symbol. These are the two parameters which it takes. Our constructor does not have any, hence the empty body. The mappings are used to keep track of information such as whether a metadata hash has been minted, meaning that we prevent the minting of that metadata again and also to map the token id to an **Item**.

- The **createCollectible(string memory metadata, uint256 royalty)** function:

```
function createCollectible(string memory metadata, uint256 royalty) public returns (uint256)
{
    require(
        !hasBeenMinted[metadata],
        "This metadata has already been used to mint an NFT."
    );
    require(
        royalty >= 0 && royalty <= 40,
        "Royalties must be between 0% and 40%"
    );
    Item memory newItem = Item(msg.sender, msg.sender, royalty);
    items.push(newItem);
    uint256 newItemId = items.length;
    _safeMint(msg.sender, newItemId);
    _setTokenURI(newItemId, metadata);
    tokenIdToItem[newItemId] = newItem;
    hasBeenMinted[metadata] = true;
    emit ItemMinted(newItemId, msg.sender, metadata, royalty);
    return newItemId;
}
```

The function takes two parameters - metadata and royalty. In the beginning of the function body we see a couple of guard conditions which are used to prevent unwanted transaction execution and to revert the transaction if the conditions are not fulfilled. We use the mapping which we have defined above to check whether the metadata has been minted. We also check whether the royalty is between 0% and 40%. Should we pass these conditions, we can now move on to creating our **Item** with the information we have. At this point the creator is both the owner and the creator, so we use **msg.sender** which is one of Solidity's global variables and denotes the caller of the function. Our third property is the royalty. After we push this **Item** to the array, we make use of the functions which the [ERC721URIStorage.sol](#) provides us, namely:

```
_safeMint(msg.sender, newItemId);
_setTokenURI(newItemId, metadata);
```

This will do the minting for us and associate the item id (token id) with the metadata which we have provided. At the end we update the mappings accordingly with the new information and return the token id.

- Furthermore, we define a couple of view functions. These do not cost any gas, since we do not change the state of the blockchain by calling them. In the second function you can notice that in the **returns** part we do not have **Item**, but rather the properties of the **Item**. Solidity allows returning multiple values.

```
function getItemsLength() public view returns (uint256) {
    return items.length;
}

function getItem(uint256 tokenId) public view returns (address, address, uint256)
{
    return (tokenIdToItem[tokenId].owner, tokenIdToItem[tokenId].creator, tokenIdToItem[tokenId].royalty);
}
}
```

Creating Our NFT Marketplace.sol Contract

- Inside the **contracts/** folder, we create a new [Marketplace.sol](#) file again with the necessary functionality. This might look slightly more complicated but once again, I will go through each line of code, so that at the end you can make sense of the logic entirely.

- [Marketplace.sol](#) logic

- At the top we define as usual the solidity version:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.6;
```

- Next we will import our already created [Collectible.sol](#) contract and inherit from it, since we would want to make use of some of the public state variables there:

```
import './Collectible.sol';

contract Marketplace is Collectible {
    using SafeMath for uint256;
```

Note: You might have noticed that we again use **SafeMath** for the **uint256**. This is to prevent the overflows that might happen.

- Afterwards we will define the state variables, events and modifiers:

```
struct Listing {
    uint256 price;
    address owner;
}

mapping (uint256 => Listing) public tokenIdToListing;
mapping (uint256 => bool) public hasBeenListed;
mapping (uint256 => address) public claimableByAccount;
event ItemListed(uint256 tokenId, uint256 price, address seller);
event ListingCancelled(uint256 tokenId, uint256 price, address seller);
event ItemBought(uint256 tokenId, uint256 price, address buyer);

modifier onlyTokenOwner(uint256 tokenId) {
    require(
        msg.sender == ownerOf(tokenId),
        "Only the owner of the token id can call this function."
    );
}

modifier onlyListingAccount(uint256 tokenId) {
    require(
        msg.sender == claimableByAccount[tokenId],
        "Only the address that has listed the token can cancel the listing."
    );
}
```

You can see that again we have a struct for the **Listing** of a NFT. We use it to define who has listed a NFT and for what price. We have again some mappings to keep track of vital information such as to which token id a listing belongs to, whether the token id has been listed, as we do not want any double listings of the same NFT and also we have a mapping to keep track of the address that can claim the NFT after it has been listed. This is of course the owner of the item. We need this, because you will see in a bit that we transfer the NFT to the smart contract when listing it, thus making the smart contract the new owner. We have some events for the different functions, namely for listing, cancelling a listing and buying an item. The new concepts which you see here are the modifiers. These can be appended as function modifiers to the functions and act exactly the same as the **require()** statements. They are usually used to prevent writing the same conditions for different functions or restricting access only to specific addresses. In this case we have two modifiers. The first one is used to prevent other addresses from listing a token which they do not own, whereas the second one is used for allowing only the address that has listed the token to cancel the listing.

- The **listItem(uint256 tokenId, uint256 price)** function:

```

function listItem(uint256 tokenId, uint256 price) public onlyTokenOwner(tokenId)
{
    require(!hasBeenListed[tokenId], "The token can only be listed once");
    _transfer(msg.sender, address(this), tokenId);
    claimableByAccount[tokenId] = msg.sender;
    tokenIdToListing[tokenId] = Listing(
        price,
        msg.sender
    );
    hasBeenListed[tokenId] = true;
    emit ItemListed(tokenId, price, msg.sender);
}

```

The function takes two parameters, namely the token id and the price. We begin by defining the constraints which are that only the token owner can list the NFT and that this NFT has not been listed already. Then we proceed by using the `_transfer(msg.sender, address(this), tokenId)` function, which is provided by the **ERC721.sol** contract, and we transfer the token to the **Marketplace.sol** contract. Then we specify the **msg.sender** as the address that can cancel the listing and we update the mappings accordingly by creating a new Listing and by specifying that the token id has been listed. At the end, as usual, we emit an event.

- The **cancelListing(uint256 tokenId)** function:

```

function cancelListing(uint256 tokenId) public onlyListingAccount(tokenId)
{
    _transfer(address(this), msg.sender, tokenId);
    uint256 price = tokenIdToListing[tokenId].price;
    delete claimableByAccount[tokenId];
    delete tokenIdToListing[tokenId];
    delete hasBeenListed[tokenId];
    emit ListingCancelled(tokenId, price, msg.sender);
}

```

Here our constraint is that only the address that has listed the item can cancel the listing. Since the mapping `claimableByAccount[tokenId]` is then cleared via the `delete` keyword we do not need to check that the item has been listed. Here we transfer the item from the Marketplace smart contract back to the one who listed it, clear the mappings and emit an event by providing it information about the token id, the price and who cancelled the listing.

- The **buyItem(uint256 tokenId)** function:

```

function buyItem(uint256 tokenId) public payable {
    require(hasBeenListed[tokenId], "The token needs to be listed in order to be bought.");
    require(tokenIdToListing[tokenId].price == msg.value, "You need to pay the correct price.");

    //split up the price between owner and creator
    uint256 royaltyForCreator = tokenIdToItem[tokenId].royalty.mul(msg.value).div(100);
    uint256 remainder = msg.value.sub(royaltyForCreator);
    //send to creator
    (bool isRoyaltySent, ) = tokenIdToItem[tokenId].creator.call{value: royaltyForCreator}("");
    require(isRoyaltySent, "Failed to send AVAX");
    //send to owner
    (bool isRemainderSent, ) = tokenIdToItem[tokenId].owner.call{value: remainder}("");
    require(isRemainderSent, "Failed to send AVAX");

    //transfer the token from the smart contract back to the buyer
    _transfer(address(this), msg.sender, tokenId);

    //Modify the owner property of the item to be the buyer
    Item storage item = tokenIdToItem[tokenId];
    item.owner = msg.sender;

    //clean up
    delete tokenIdToListing[tokenId];
    delete claimableByAccount[tokenId];
    delete hasBeenListed[tokenId];
    emit ItemBought(tokenId, msg.value, msg.sender);
}

```

Here the function takes the token id as a parameter and is a **payable** function, meaning that the user can send AVAX via it to the smart contract. We then first check whether the item has been listed and whether the **msg.value** which we send with our function call equals the price of the token. If that is the case we split up the **msg.value** based on the royalty that is defined in the **Item**. **msg.value** is another global variable in Solidity:

```

uint256 royaltyForCreator = tokenIdToItem[tokenId].royalty.mul(msg.value).div(100);
uint256 remainder = msg.value.sub(royaltyForCreator);

```

In the first line we multiply the royalty by the **msg.value** and then divide it by 100, since we are talking about percentages. Meaning that if the buyer pays 10 AVAX for the NFT and the royalty is 20%, 2 AVAX would go to the creator and the remaining 8 AVAX would go to the seller. This happens in the next lines:

```
(bool isRoyaltySent, ) = tokenIdToItem[tokenId].creator.call{value: royaltyForCreator}("");
require(isRoyaltySent, "Failed to send AVAX");
(bool isRemainderSent, ) = tokenIdToItem[tokenId].owner.call{value: remainder}("");
require(isRemainderSent, "Failed to send AVAX");
```

Afterwards we transfer the NFT from the Marketplace smart contract to the buyer and update the **Item** by modifying the **owner** property. Finally, as we did before, we clean up the mappings and emit an event passing the necessary information to it.

- At the end we define a view function which is used to obtain information about a certain listing. Again, calling this function costs no gas.

```
function getListing(uint256 tokenId) public view returns (uint256, address)
{
    return (tokenIdToListing[tokenId].price, tokenIdToListing[tokenId].owner);
}
```

Testing Our Smart Contracts

Now that we have finished writing our smart contracts it is very important that we test them thoroughly for erroneous behaviour. As we know, once deployed on the blockchain, they are immutable. We will be using [Mocha.js](#) for testing our contracts. It is integrated into the Truffle framework, so we do not need to install it. We do, however, need [Open Zeppelin's Test Helpers](#), so we will import these in our tests. With that in mind, let us quickly jump into the **test**/ directory of our project root and inside of it create two files, namely:

- [collectible.test.ts](#)
- [marketplace.test.ts](#)

I. Let us start with the first one:

As you may notice most of the actions are repetitive, but will note down some key components which you would need in order to understand how to approach some often met cases.

1. First, we import our dependencies at the top and then we define the scope of our test:

```
const Collectible = artifacts.require('./Collectible')
import { expectRevert } from '@openzeppelin/test-helpers'

contract('Collectible', ([contractDeployer, creator, buyer]) => {
```

As you can see we import the **Collectible** contract and also an `expectRevert` function that will help us with checking whether the functions revert correctly upon false input. Afterwards, we define the scope of our test. Truffle uses the `contract()` function instead of Mocha's `describe()` function. The differences are minimal, so think of them as identical. If you are curious, check out the Truffle docs about the [explanation](#). The first parameter is the contract name, the second is simply a list of addresses. In this case Truffle's testing environment provides us with 10 accounts, each funded with 100 ETH. Yes, the Truffle's testing environment is based on Ethereum, but since the C-Chain is [EVM](#) compatible, think of them as 100 AVAX. Of course, these funds are not real ones and serve only for testing purposes.

2. Afterwards, we define a `before()` hook. This hook runs before our tests and we can use it to deploy the contract.

```
let collectible;

before(async () => {
    collectible = await Collectible.new({ from: contractDeployer })
});
```

Note: We do not actually need `{from: contractDeployer}` as a parameter. If it is missing, Truffle would automatically take the first address into consideration for the function call. However, you will see that for minting an NFT we would use the **creator** address, so we would then need to specify this. Each function of our contracts would be part of a `describe()` block. That way we can structure our tests efficiently, so that we can find our way easier.

3. In the first `describe()` block we test whether our contract was deployed correctly. For that we need to write individual tests or `it()`.

```
describe('Collectible deployment', async () => {
    it('Deploys the Collectible SC successfully.', async () => {
        console.log('Address is ', collectible.address)
        assert.notEqual(collectible.address, '', 'should not be empty');
        assert.notEqual(collectible.address, 0x0, 'should not be the 0x0 address');
        assert.notEqual(collectible.address, null, 'should not be null');
        assert.notEqual(collectible.address, undefined, 'should not be undefined');
    })
}

it('The collectible SC should have a name and a symbol.', async () => {
    const name = await collectible.name()
    assert.equal(name, 'NFTCollectible', 'The name should be NFTCollectible.')
    const symbol = await collectible.symbol()
    assert.equal(symbol, 'NFTC', 'The symbol should be NFTC.')
})
```

As you can see an `it()` function takes as parameters a description of the test and an asynchronous function. In the first case, we check whether the `collectible.address` is not equal to those illegal values. In the second test we check whether our `Collectible.sol` has a name and a symbol. Since those variables are public in the `ERC721.sol` implementation we can call them as getters. The value is stored in a variable and then this variable is compared to the expected name. If you jump to the `Collectible.sol` file you can see the expected name in the `constructor()`.

4. In the second `describe()` block we test our `createCollectible()` function. For that we need to write individual tests for every statement that we make.

```
describe('Mint an NFT and set a royalty.', async () => {

    it('The hash \'metadata\' is not minted before the function call.', async () => {
        const hasBeenMinted = await collectible.hasBeenMinted('metadata')
        assert.equal(hasBeenMinted, false, 'The hash \'metadata\' has not been minted, so it should be false.')
    })

    it('The royalty needs to be a number between 0 and 40.', async () => {
        await expectRevert(collectible.createCollectible('metadata', 41), "Royalties must be between 0% and 40%.");
    })

    it('Give a new id to a newly created token', async () => {
        const newTokenId = await collectible.createCollectible.call('metadata', 20, { from: creator })
        assert.equal(parseInt(newTokenId.toString()), 1, 'The new token id should be 1.')
    })

    it('Mint a NFT and emit events.', async () => {
        const result = await collectible.createCollectible('metadata', 20, { from: creator })
        assert.equal(result.logs.length, 2, 'Should trigger two events.');
        //event Transfer
        assert.equal(result.logs[0].event, 'Transfer', 'Should be the \'Transfer\' event.');
        assert.equal(result.logs[0].args.from, '0x0', 'Should be the 0x0 address.');
        assert.equal(result.logs[0].args.to, creator, 'Should log the recipient which is the creator.');
        assert.equal(result.logs[0].args tokenId, 1, 'Should log the token id which is 1.');

        //event ItemMinted
        assert.equal(result.logs[1].event, 'ItemMinted', 'Should be the \'ItemMinted\' event.');
        assert.equal(result.logs[1].args tokenId, 1, 'Should be the token id 1.');
        assert.equal(result.logs[1].args.creator, creator, 'Should log the creator.');
        assert.equal(result.logs[1].args.metadata, 'metadata', 'Should log the metadata correctly.');
        assert.equal(result.logs[1].args.royalty, 20, 'Should log the royalty as 20.');
    })

    it('The items array has a length of 1.', async () => {
        const itemsLength = await collectible.getItemsLength()
        assert.equal(itemsLength, 1, 'The items array should have 1 entry in it.')
    })

    it('The new item has the correct data.', async () => {
        const item = await collectible.getItem(1)
        assert.notEqual(item['0'], buyer, 'The buyer should not be the creator.')
        assert.equal(item['0'], creator, 'The creator is the owner.')
        assert.equal(item['1'], creator, 'The creator is the creator.')
        assert.equal(item['2'], 20, 'The royalty is set to 20.')
    })

    it('Check if hash has been minted and that you cannot mint the same hash again.', async () => {
        const hasBeenMinted = await collectible.hasBeenMinted('metadata')
        assert.equal(hasBeenMinted, true, 'The hash \'metadata\' has been minted.')
        await expectRevert(collectible.createCollectible('metadata', 30, { from: creator }), 'This metadata has already been used to mint an NFT.');
    })
})
})
```

- First we check that the `'metadata'` is not minted. For that we call the `hasBeenMinted('metadata')` function which is in fact our mapping in our `Collectible.sol` file. This returns us a boolean which is false.
- Afterwards, we expect that the minting function reverts if we provide a royalty that is not between 0% and 40%. In that case we try with 41%.
- Then, before we complete the transaction we can check what the return value would be. As we know, our `createCollectible()` function returns a token id. We can grab this by executing the function without changing the state:

```
it('Give a new id to a newly created token', async () => {
    const newTokenId = await collectible.createCollectible.call('metadata', 20, { from: creator })
    assert.equal(parseInt(newTokenId.toString()), 1, 'The new token id should be 1.')
})
```

Then we simply compare the `newTokenId` to 1 and expect them to be equal, since our first NFT should have the token id 1.

- Now we do not only execute the `createCollectible()` function but also change the state in our next `it()`:

```

it('Mint a NFT and emit events.', async () => {
    const result = await collectible.createCollectible('metadata', 20, { from: creator })
    assert.equal(result.logs.length, 2, 'Should trigger two events.');
    //event Transfer
    assert.equal(result.logs[0].event, 'Transfer', 'Should be the \'Transfer\' event.');
    assert.equal(result.logs[0].args.from, '0x0', 'Should be the 0x0 address.');
    assert.equal(result.logs[0].args.to, creator, 'Should log the recipient which is the creator.');
    assert.equal(result.logs[0].args tokenId, 1, 'Should log the token id which is 1.');

    //event ItemMinted
    assert.equal(result.logs[1].event, 'ItemMinted', 'Should be the \'ItemMinted\' event.');
    assert.equal(result.logs[1].args tokenId, 1, 'Should be the token id 1.');
    assert.equal(result.logs[1].args.creator, creator, 'Should log the creator.');
    assert.equal(result.logs[1].args.metadata, 'metadata', 'Should log the metadata correctly.');
    assert.equal(result.logs[1].args.royalty, 20, 'Should log the royalty as 20.');
})
)

```

Our variable which is the result of the function call is no longer the token id, but a transaction receipt, meaning that we obtain much more information out of it. Cool, right? Let us put this information to use. We test whether the correct events are emitted since they are the signal that we need. In this case we have two. One is the *Transfer* event which comes from the `_safeMint(msg.sender, newItemId)` function of the `ERC721.sol` smart contract. The other one is our own *ItemMinted* event. We check for the correct name and the correct arguments.

- In the remaining `it()`-s we check whether the mappings were updated accordingly and whether our `Item` has the correct values. Our final `it()` makes sure that the transaction reverts if we call the `createCollectible()` function with the same metadata parameter value.

5. Now that we are done writing the test, in our console we simply run the command:

```
npx truffle test
```

Note: You might notice that this would run the command `truffle compile` beforehand. This would create a **build/contracts** folder in our root directory where the `.json` representations of all of our used contracts are stored. These are in fact used when you call functions on the frontend.

II. Go inside the `marketplace.test.ts` file and have a look at the code. As you may notice, most of the concepts such as testing the deployment, function reverts and emitted events are repeated here, so I will only go through the differences:

1. Once again, at the top we are importing the `Marketplace` contract as well as some helping functions:

```

const Marketplace = artifacts.require('./Marketplace')
const { toBN } = web3.utils
import { expectRevert, BN } from '@openzeppelin/test-helpers'
import { convertTokensToWei } from '../utils/tokens'

```

We use the `toBN()` function to convert the balances of the addresses, which are returned as strings, to Big Numbers, so that we can perform an adding. We also import a function `convertTokenToWei` from a `utils/` folder of our project's root which we do not have yet, so let us create it and inside of it create a `tokens.ts` file. Then copy the code below in there:

```

export const convertTokensToWei = (n) => {
    return web3.utils.toWei(n, 'ether')
}

module.exports = { convertTokensToWei }

```

We use this function, so that we do not have to write 18 zeroes after the AVAX amount that a buyer would pay for a NFT. In reality, transferring 5 AVAX means that we transfer 500000000000000000000000 as a value. In order to not have to write all those zeroes, we can simply call `convertTokensToWei('5')` and the function will add the zeroes for us. Now, back to our `marketplace.test.ts` test script.

2. We create a `before()` hook. There we deploy the marketplace contract and we also mint a NFT. Notice again that our `Marketplace.sol` has all the functions which `Collectible.sol` has, hence we can call the `createCollectible()` function:

```

before(async () => {
    marketplace = await Marketplace.new({ from: contractDeployer })
    await marketplace.createCollectible('metadata', 20, { from: creator })
});

```

In this case the creator address is the one who calls the function and therefore is the owner of the first NFT.

3. Afterwards, we simply test every statement which we make inside the function just like we did for the `collectible.test.ts`.

4. We do the same for the `cancelListing()` function.

5. For the `buyItem()` function things are not that different. The interesting part is the final `it()` where we check the balances of the seller, buyer and creator after the second sale, since for the first one the seller is equal to the creator:

```

it('The balances of creator, first buyer who is now the seller and second buyer are correct.', async () => {
    //List the item again, only this time by the new owner
    await marketplace.listItem(1, convertTokensToWei('10'), { from: buyer })
}
)

```

```

    const balanceOfBuyerBeforePurchase = await web3.eth.getBalance(buyer)
    const balanceOfCreatorBeforePurchase = await web3.eth.getBalance(creator)
    const balanceOfSecondBuyerBeforePurchase = await web3.eth.getBalance(secondBuyer)
    await marketplace.buyItem(1, { from: secondBuyer, value: convertTokensToWei('10') })
    const balanceOfSecondBuyerAfterPurchase = await web3.eth.getBalance(secondBuyer)
    const isCorrectSecondBuyerBalanceDifference =
    toBN(balanceOfSecondBuyerAfterPurchase).add(toBN(convertTokensToWei('10'))).lt(toBN(balanceOfSecondBuyerBeforePurchase))
    assert.equal(isCorrectSecondBuyerBalanceDifference, true, 'The balance of the second buyer should decrease by
10 AVAX plus gas paid.')
    const balanceOfBuyerAfterPurchase = await web3.eth.getBalance(buyer)
    const balanceOfCreatorAfterPurchase = await web3.eth.getBalance(creator)
    assert.equal(balanceOfBuyerAfterPurchase,
    toBN(balanceOfBuyerBeforePurchase).add(toBN(convertTokensToWei('10')).mul(new BN('80')).div(new BN('100'))), 'The balance
of the seller should increase by 80% of the sold amount.')
    assert.equal(balanceOfCreatorAfterPurchase,
    toBN(balanceOfCreatorBeforePurchase).add(toBN(convertTokensToWei('10')).mul(new BN('20')).div(new BN('100'))), 'The
balance of the creator should increase by 20% of the sold amount.')
})

```

What we do here is that the new owner lists the item for 10 AVAX. He is denoted by the **buyer** address, since he has previously bought the NFT from the creator. A `secondBuyer` purchases the item, so we check whether the `secondBuyer` has at least 10 AVAX less, because we need to take into account the gas paid:

```

const balanceOfSecondBuyerBeforePurchase = await web3.eth.getBalance(secondBuyer)
await marketplace.buyItem(1, { from: secondBuyer, value: convertTokensToWei('10') })
const balanceOfSecondBuyerAfterPurchase = await web3.eth.getBalance(secondBuyer)
const isCorrectSecondBuyerBalanceDifference =
toBN(balanceOfSecondBuyerAfterPurchase).add(toBN(convertTokensToWei('10'))).lt(toBN(balanceOfSecondBuyerBeforePurchase))
assert.equal(isCorrectSecondBuyerBalanceDifference, true, 'The balance of the second buyer should decrease by
10 AVAX plus gas paid.')

```

For the **creator** we also check their balance before and after the purchase and that it needs to be larger by 20% of the purchase amount. The seller which is in this case the **buyer** address should get 80% of the price:

```

const balanceOfBuyerAfterPurchase = await web3.eth.getBalance(buyer)
const balanceOfCreatorAfterPurchase = await web3.eth.getBalance(creator)
assert.equal(balanceOfBuyerAfterPurchase, toBN(balanceOfBuyerBeforePurchase).add(toBN(convertTokensToWei('10')).mul(new
BN('80')).div(new BN('100'))), 'The balance of the seller should increase by 80% of the sold amount.')
assert.equal(balanceOfCreatorAfterPurchase,
toBN(balanceOfCreatorBeforePurchase).add(toBN(convertTokensToWei('10')).mul(new BN('20')).div(new BN('100'))), 'The
balance of the creator should increase by 20% of the sold amount.')

```

6. Now that we are done writing the test, in our console we simply run the command:

```
npx truffle test
```

7. To check the test coverage we can run the command:

```
truffle run coverage
```

Deploying Our Smart Contracts

1. Now that our contracts have passed the tests, let us have a look at the sub-dir **migrations/** folder which Truffle provided us in the beginning. Inside of it we have the [1_initial_migration.ts](#) script which is used to deploy the **Migrations.sol** contract that is available in the **contracts/** folder. This contract simply keeps track of the migrations that we do. In order to migrate our own contracts, we create another script called [2_deploy_contracts.ts](#) and inside of it paste the following lines of code:

```

const Collectible = artifacts.require('Collectible')
const Marketplace = artifacts.require('Marketplace')

module.exports = function (deployer) {
  deployer.deploy(Collectible)
  deployer.deploy(Marketplace)
} as Truffle.Migration

// because of https://stackoverflow.com/questions/40900791/cannot-redeclare-block-scoped-variable-in-unrelated-files
export { }

```

As you can see, it is pretty straightforward. We import the contracts and deploy them via the `deployer` parameter. This is taken care by Truffle.

2.1. In order to deploy our contracts to the [Fuji testnet](#), all we need to do is simply run the command:

```
truffle migrate --network fuji
```

2.2. And should we want to deploy to the Avalanche C-Chain, we simply run:

```
truffle migrate --network mainnet
```

Conclusion

To sum up, in this tutorial we got familiar with the following concepts:

1. We managed to create our very own [ERC721 smart contract](#) based on the [Open Zeppelin implementation](#) utilizing the Truffle framework;
2. We extended this contract by including royalties;
3. We created a marketplace where one could list their NFTs, cancel their listings or buy other NFTs;
4. We learned how to test our smart contracts extensively using the integrated Mocha.js library in Truffle;
5. We deployed our final contracts to the Fuji testnet;

Last but not least, should you try to extend the smart contracts, here are some potential ideas:

- For the [Collectible.sol](#) you could add more special properties for your NFTs should you want to use gamification;
- For the [Marketplace.sol](#) you could implement the option to bid on an item;
- Include an expiration date for a listing on the marketplace;

Or literally anything else that comes to your mind. The opportunities are limitless :)

Create a Chat Dapp Using Solidity and ReactJS

Introduction

In this tutorial we will build a decentralized chat application on Avalanche's Fuji test-network from scratch. The dapp will allow users to connect with other people and chat with them. We will develop our smart contract using Solidity which will be deployed on Avalanche's C-chain. We will have a basic, easy-to-use UI developed using ReactJS. So, let us begin!

Requirements

- Basic familiarity with [ReactJS](#) and [Solidity](#)
- [Node.js](#) v10.18.0+
- [MetaMask extension](#) on your browser

Implementing the Smart Contract

Our chat dapp needs the basic functionality allowing users to connect with and share messages with friends. To accomplish this, we will write the functions responsible for creating an account, adding friends and sending messages.

Account Creation

We will define 3 functions :

- The `checkUserExists(pubkey)` function is used to check if a user is registered with our application or not. It will help make sure duplicate users are not created and it will also be called from other functions to check their existence.
- The `createAccount(username)` function registers a new user on the platform with the provided username.
- The `getUsername(pubkey)` function will return the username of the given user if it exists.

Adding Friends

Here also we will define 3 functions :

- The `checkAlreadyFriends(pubkey1, pubkey2)` function checks whether two users are already friends with each other or not. This is needed to prevent duplicate channel between the same parties and will also be used to prevent a user from sending messages to other users unless they are friends.
- The `addFriend(pubkey, name)` function mark the two users as friend if they both are registered on the platform and are already not friends with each other.
- The `getMyFriendList()` function will return an array of friends of the given user.

Messaging

The final part of the Solidity contract will enable the exchange of messages between users. We will divide the task into two functions `sendMessage()` and `readMessage()`.

- The `sendMessage()` function allows a user to send messages to another registered user (friend). This is done with `checkUserExists(pubkey)` and `checkAlreadyFriends(pubkey1, pubkey2)`.
- The `readMessage()` function returns the chat history that has happened between the two users so far.

User Data Collections

We will have three types of user-defined data :

- `user` will have the properties `name` which stores the username, and `friendList` which is an array of other users.
- `friend` will have the properties `pubkey` which is the friends' public address, and `name` which the user would like to refer them as.
- `message` has three properties: `sender`, `timestamp` and `msg`, which is short for "message."

We would maintain 2 collections in our database:

- `userList` where all the users on the platform are mapped with their public address.
- `allMessages` stores the messages. As Solidity does not allow user-defined keys in a mapping, we can instead hash the public keys of the two users. This value can then be stored in the mapping.

Deploying the Smart Contract

Setting up MetaMask

Log in to MetaMask -> Click the Network drop-down -> Select Custom RPC



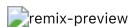
Fuji Testnet Settings:

- **Network Name:** Avalanche Fuji C-Chain
- **New RPC URL:** <https://api.avax-test.network/ext/bc/C/rpc>
- **ChainID:** 43113
- **Symbol:** C-AVAX
- **Explorer:** <https://testnet.snowtrace.io>

Find your address from the given [faucet](#).

Deploy Using Remix

Open [Remix](#) -> Select Solidity



Create a `Database.sol` file in the Remix file explorer, and paste the following code :

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract Database {

    // Stores the default name of an user and her friends info
    struct user {
        string name;
        friend[] friendList;
    }

    // Each friend is identified by its address and name assigned by the second party
    struct friend {
        address pubkey;
        string name;
    }

    // message construct stores the single chat message and its metadata
    struct message {
        address sender;
        uint256 timestamp;
        string msg;
    }

    // Collection of users registered on the application
    mapping(address => user) userList;
    // Collection of messages communicated in a channel between two users
    mapping(bytes32 => message[]) allMessages; // key : Hash(user1,user2)

    // It checks whether a user(identified by its public key)
    // has created an account on this application or not
    function checkUserExists(address pubkey) public view returns(bool) {
        return bytes(userList[pubkey].name).length > 0;
    }

    // Registers the caller(msg.sender) to our app with a non-empty username
    function createAccount(string calldata name) external {

```

```

require(checkUserExists(msg.sender)==false, "User already exists!");
require(bytes(name).length>0, "Username cannot be empty!");
userList[msg.sender].name = name;
}

// Returns the default name provided by an user
function getUsername(address pubkey) external view returns(string memory) {
    require(checkUserExists(pubkey), "User is not registered!");
    return userList[pubkey].name;
}

// Adds new user as your friend with an associated nickname
function addFriend(address friend_key, string calldata name) external {
    require(checkUserExists(msg.sender), "Create an account first!");
    require(checkUserExists(friend_key), "User is not registered!");
    require(msg.sender!=friend_key, "Users cannot add themselves as friends!");
    require(checkAlreadyFriends(msg.sender,friend_key)==false, "These users are already friends!");

    _addFriend(msg.sender, friend_key, name);
    _addFriend(friend_key, msg.sender, userList[msg.sender].name);
}

// Checks if two users are already friends or not
function checkAlreadyFriends(address pubkey1, address pubkey2) internal view returns(bool) {

    if(userList[pubkey1].friendList.length > userList[pubkey2].friendList.length)
    {
        address tmp = pubkey1;
        pubkey1 = pubkey2;
        pubkey2 = tmp;
    }

    for(uint i=0; i<userList[pubkey1].friendList.length; ++i)
    {
        if(userList[pubkey1].friendList[i].pubkey == pubkey2)
            return true;
    }
    return false;
}

// A helper function to update the friendList
function _addFriend(address me, address friend_key, string memory name) internal {
    friend memory newFriend = friend(friend_key, name);
    userList[me].friendList.push(newFriend);
}

// Returns list of friends of the sender
function getMyFriendList() external view returns(friend[] memory) {
    return userList[msg.sender].friendList;
}

// Returns a unique code for the channel created between the two users
// Hash(key1,key2) where key1 is lexicographically smaller than key2
function _getChatCode(address pubkey1, address pubkey2) internal pure returns(bytes32) {
    if(pubkey1 < pubkey2)
        return keccak256(abi.encodePacked(pubkey1, pubkey2));
    else
        return keccak256(abi.encodePacked(pubkey2, pubkey1));
}

// Sends a new message to a given friend
function sendMessage(address friend_key, string calldata _msg) external {
    require(checkUserExists(msg.sender), "Create an account first!");
    require(checkUserExists(friend_key), "User is not registered!");
    require(checkAlreadyFriends(msg.sender,friend_key), "You are not friends with the given user");

    bytes32 chatCode = _getChatCode(msg.sender, friend_key);
    message memory newMsg = message(msg.sender, block.timestamp, _msg);
    allMessages[chatCode].push(newMsg);
}

// Returns all the chat messages communicated in a channel
function readMessage(address friend_key) external view returns(message[] memory) {
    bytes32 chatCode = _getChatCode(msg.sender, friend_key);
    return allMessages[chatCode];
}
}

```

Navigate to the Solidity compiler Tab on the left side navigation bar and click the blue button to compile the `Database.sol` contract. Note down the `ABI` as it will be required in the next section.

Navigate to Deploy Tab and open the "ENVIRONMENT" drop-down. Select "Injected Web3" (make sure MetaMask is loaded) and click "Deploy" button.

Approve the transaction on MetaMask pop-up interface. Once our contract is deployed successfully, Note down the `contract address`.

An Application Binary Interface (ABI) is a JSON object which stores the metadata about the methods of a contract like data type of input parameters, return data type & property of the method like payable, view, pure etc. You can learn more about the ABI from the [solidity documentation](#).

Creating a Frontend in React

Now, we are going to create a React app scaffold and set up the frontend of the application.

Open a terminal and navigate to the directory where we will create the application.

```
cd /path/to/directory
```

Now use `npm` to install `create-react-app`. `-g` flag denotes that the package should be installed globally.

```
npm install -g create-react-app
```

Create a new react app.

```
create-react-app avalanche-chat-app
```

Move to the newly created directory and install the given dependencies.

```
cd avalanche-chat-app
npm install --save ethers@5.1.4 react-bootstrap@1.5.2 bootstrap@4.6.0
```

Open `index.html` file in the `public` directory, and paste the following HTML :

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <title>Chat dapp</title>
</head>

<body>
  <div id="root"></div>
</body>

</html>
```

Move out of the public directory and create a new directory `components` inside `src` directory, where we will be keeping all our React components, using the following command :

```
mkdir ./src/components
cd ./src/components
```

Now let's create the component having the navbar of our dapp. Create a new file called `NavBar.jsx` and paste the following code :

```
import React from "react";
import { Button, Navbar } from "react-bootstrap";

// This component renders the Navbar of our application
export function NavBar( props ){
  return (
    <Navbar bg="dark" variant="dark">
      <Navbar.Brand href="#home">
        Avalanche Chat App
      </Navbar.Brand>
      <Navbar.Toggle />
      <Navbar.Collapse className="justify-content-end">
        <Navbar.Text>
          <Button style={{ display: props.showButton }} variant="success" onClick={ async () => { props.login(); } }>
            Connect to MetaMask
          </Button>
        <div style={{ display: props.showButton === "none" ? "block" : "none" }}>
          Signed in as:
        </div>
      </Navbar.Collapse>
    </Navbar>
  );
}
```

```

        <a href="#">
            { props.username }
        </a>
    </div>
</Navbar.Text>
</Navbar.Collapse>
</Navbar>
);
}

```

All the contacts will have a card with their name and public key. Create a new file called `ChatCard.jsx` and paste the following code :

```

import React from "react";
import { Row, Card } from "react-bootstrap";
import 'bootstrap/dist/css/bootstrap.min.css';

// This is a function which renders the friends in the friends list
export function ChatCard( props ){
    return (
        <Row style={{ marginRight:"0px" }}>
            <Card border="success" style={{ width:'100%', alignSelf:'center', marginLeft:"15px" }} onClick={ () => {
                props.getMessages( props.publicKey );
            }}>
                <Card.Body>
                    <Card.Title> { props.name } </Card.Title>
                    <Card.Subtitle> { props.publicKey.length > 20 ? props.publicKey.substring(0, 20) + " ..." : props.publicKey } </Card.Subtitle>
                </Card.Body>
            </Card>
        </Row>
    );
}

```

Each message will be rendered by the Message component. This component will have the timestamp , senders' name and the message. Create a new file called `Message.jsx` and paste the following code :

```

import React from "react";
import { Row, Card } from "react-bootstrap";

// This is a functional component which renders the individual messages
export function Message( props ){
    return (
        <Row style={{ marginRight:"0px" }}>
            <Card border="success" style={{ width:'80%', alignSelf:'center', margin:"0 0 5px " + props.marginLeft, float:"right", right:"0px" }}>
                <Card.Body>
                    <h6 style={{ float:"right" }}>
                        { props.timeStamp }
                    </h6>
                    <Card.Subtitle>
                        <b>
                            { props.sender }
                        </b>
                    </Card.Subtitle>
                    <Card.Text>
                        { props.data }
                    </Card.Text>
                </Card.Body>
            </Card>
        </Row>
    );
}

```

To add a new contact we will make AddNewChat component. It will show a modal on clicking the NewChat button and ask for the contact details. Create a new file called `AddNewChat.jsx` and paste the following code :

```

import React from "react";
import { useState } from "react";
import { Button, Modal, Form } from "react-bootstrap";

// This Modal help Add a new friend
export function AddNewChat( props ){
    const [show, setShow] = useState( false );
    const handleClose = () => setShow( false );
    const handleShow = () => setShow( true );
    return (
        <div className="AddNewChat" style={{ position:"absolute", bottom:"0px", padding:"10px 45px 0 45px", margin:"0 95px 0 0", >
    );
}

```

```

width:"97%" }}>
    <Button variant="success" className="mb-2" onClick={ handleShow }>
        + NewChat
    </Button>
    <Modal show={ show } onHide={ handleClose }>
        <Modal.Header closeButton>
            <Modal.Title> Add New Friend </Modal.Title>
        </Modal.Header>
        <Modal.Body>
            <Form.Group>
                <Form.Control required id="addPublicKey" size="text" type="text" placeholder="Enter Friends Public Key" />
            <br />
                <Form.Control required id="addName" size="text" type="text" placeholder="Name" />
            <br />
            </Form.Group>
        </Modal.Body>
        <Modal.Footer>
            <Button variant="secondary" onClick={ handleClose }>
                Close
            </Button>
            <Button variant="primary" onClick={ () => {
                props.addHandler( document.getElementById('addName').value,
document.getElementById('addPublicKey').value );
                handleClose();
            }}>
                Add Friend
            </Button>
        </Modal.Footer>
    </Modal>
</div>

};

}

```

Now lets create a new file called `Components.js` and export all the components together. Paste the following code :

```

export { NavBar } from "./NavBar";
export { AddNewChat } from "./AddNewChat";
export { Message } from "./Message";
export { ChatCard } from "./ChatCard";

```

Move out to the `src` directory and create a new file called `App.jsx` and paste the following code :

```

import React from "react";
import { useState, useEffect } from "react";
import { Container, Row, Col, Card, Form, Button } from 'react-bootstrap';
import { NavBar, ChatCard, Message, AddNewChat } from './components/Components.js';
import { ethers } from "ethers";
import { abi } from "./abi";

// Add the contract address inside the quotes
const CONTRACT_ADDRESS = "";

export function App( props ) {
    const [friends, setFriends] = useState(null);
    const [myName, setMyName] = useState(null);
    const [myPublicKey, setMyPublicKey] = useState(null);
    const [activeChat, setActiveChat] = useState({ friendname: null, publicKey: null });
    const [activeChatMessages, setActiveChatMessages] = useState(null);
    const [showConnectButton, setShowConnectButton] = useState("block");
    const [myContract, setMyContract] = useState(null);

    // Save the contents of abi in a variable
    const contractABI = abi;
    let provider;
    let signer;

    // Login to MetaMask and check the if the user exists else creates one
    async function login() {
        let res = await connectToMetamask();
        if( res === true ) {
            provider = new ethers.providers.Web3Provider( window.ethereum );
            signer = provider.getSigner();
            try {
                const contract = new ethers.Contract( CONTRACT_ADDRESS, contractABI, signer );
                setMyContract( contract );
            }
        }
    }
}


```

```

        const address = await signer.getAddress();
        let present = await contract.checkUserExists( address );
        let username;
        if( present )
            username = await contract.getUsername( address );
        else {
            username = prompt('Enter a username', 'Guest');
            if( username === '' ) username = 'Guest';
            await contract.createAccount( username );
        }
        setMyName( username );
        setMyPublicKey( address );
        setShowConnectButton( "none" );
    } catch(err) {
        alert("CONTRACT_ADDRESS not set properly!");
    }
} else {
    alert("Couldn't connect to MetaMask");
}
}

// Check if the MetaMask connects
async function connectToMetamask() {
    try {
        await window.ethereum.enable();
        return true;
    } catch(err) {
        return false;
    }
}

// Add a friend to the users' Friends List
async function addChat( name, publicKey ) {
    try {
        let present = await myContract.checkUserExists( publicKey );
        if( !present ) {
            alert("Given address not found: Ask him to join the app :)");
            return;
        }
        try {
            await myContract.addFriend( publicKey, name );
            const frnd = { "name": name, "publicKey": publicKey };
            setFriends( friends.concat(frnd) );
        } catch(err) {
            alert("Friend already Added! You can't be friend with the same person twice ;P");
        }
    } catch(err) {
        alert("Invalid address!");
    }
}

// Sends message to an user
async function sendMessage( data ) {
    if( !( activeChat && activeChat.publicKey ) ) return;
    const recieverAddress = activeChat.publicKey;
    await myContract.sendMessage( recieverAddress, data );
}

// Fetch chat messages with a friend
async function getMessage( friendsPublicKey ) {
    let nickname;
    let messages = [];
    friends.forEach( ( item ) => {
        if( item.publicKey === friendsPublicKey )
            nickname = item.name;
    });
    // Get messages
    const data = await myContract.readMessage( friendsPublicKey );
    data.forEach( ( item ) => {
        const timestamp = new Date( 1000*item[1].toNumber() ).toUTCString();
        messages.push({ "publicKey": item[0], "timeStamp": timestamp, "data": item[2] });
    });
    setActiveChat({ friendname: nickname, publicKey: friendsPublicKey });
    setActiveChatMessages( messages );
}

// This executes every time page renders and when myPublicKey or myContract changes
useEffect( () => {

```

```

    async function loadFriends() {
        let friendList = [];
        // Get Friends
        try {
            const data = await myContract.getMyFriendList();
            data.forEach( item => {
                friendList.push({ "publicKey": item[0], "name": item[1] });
            })
        } catch(err) {
            friendList = null;
        }
        setFriends( friendList );
    }
    loadFriends();
}, [myPublicKey, myContract]);

// Makes Cards for each Message
const Messages = activeChatMessages ? activeChatMessages.map( message => {
    let margin = "5%";
    let sender = activeChat.friendname;
    if( message.publicKey === myPublicKey ) {
        margin = "15%";
        sender = "You";
    }
    return (
        <Message marginLeft={ margin } sender={ sender } data={ message.data } timeStamp={ message.timeStamp } />
    );
}) : null;

// Displays each card
const chats = friends ? friends.map( friend => {
    return (
        <ChatCard publicKey={ friend.publicKey } name={ friend.name } getMessages={ key => getMessage( key ) } />
    );
}) : null;

return (
    <Container style={{ padding:"0px", border:"1px solid grey" }}>
        /* This shows the navbar with connect button */
        <NavBar username={ myName } login={ async () => login() } showButton={ showConnectButton } />
        <Row>
            /* Here the friends list is shown */
            <Col style={{ paddingRight:"0px", borderRight:"2px solid #000000" }}>
                <div style={{ backgroundColor:"#DCDCDC", height:"100%", overflowY:"auto" }}>
                    <Row style={{ marginRight:"0px" }}>
                        <Card style={{ width:'100%', alignSelf:'center', marginLeft:"15px" }}>
                            <Card.Header>
                                Chats
                            </Card.Header>
                            <Card>
                                <Chats />
                            </Card>
                        </Row>
                    { chats }
                    <AddNewChat myContract={ myContract } addHandler={ ( name, publicKey ) => addChat( name, publicKey ) } />
                </div>
            </Col>
            <Col xs={ 8 } style={{ paddingLeft:"0px" }}>
                <div style={{ backgroundColor:"#DCDCDC", height:"100%" }}>
                    /* Chat header with refresh button, username and public key are rendered here */
                    <Row style={{ marginRight:"0px" }}>
                        <Card style={{ width:'100%', alignSelf:'center', margin:"0 0 5px 15px" }}>
                            <Card.Header>
                                { activeChat.friendname } : { activeChat.publicKey }
                            <Button style={{ float:"right" }} variant="warning" onClick={ () => {
                                if( activeChat && activeChat.publicKey )
                                    getMessage( activeChat.publicKey );
                            } }>
                                Refresh
                            </Button>
                            <Card.Header>
                            </Card>
                    </Row>
                    /* The messages will be shown here */
                    <div className="MessageBox" style={{ height:"400px", overflowY:"auto" }}>
                        { Messages }
                    </div>
                    /* The form with send button and message input fields */
                    <div className="SendMessage" style={{ borderTop:"2px solid black", position:"relative", bottom:"0px", width:"100%" }}>
                        <Formik
                            initialValues={ initialValues }
                            onSubmit={ ( values ) => {
                                const message = values.message;
                                const recipient = activeChat.friendname;
                                const publicKey = activeChat.publicKey;
                                const timestamp = Date.now();
                                const data = { message, recipient, publicKey, timestamp };
                                const response = await myContract.sendMessage(data);
                                if (response.status === 200) {
                                    const updatedChats = [...chats];
                                    const updatedChat = {
                                        ...activeChat,
                                        messages: [...activeChat.messages, message]
                                    };
                                    updatedChats = updatedChats.map(chat => chat.name === activeChat.name ? updatedChat : chat);
                                    setFriends(updatedChats);
                                    setMessage("");
                                }
                            } }
                        <Form />
                    </div>
                </div>
            </Col>
        </Row>
    </Container>
)

```

```

padding:"10px 45px 0 45px", margin:"0 95px 0 0", width:"97%" }>
    <Form onSubmit={ (e) => {
        e.preventDefault();
        sendMessage( document.getElementById( 'messageData' ).value );
        document.getElementById( 'messageData' ).value = "";
    }}>
        <Form.Row className="align-items-center">
            <Col xs={9}>
                <Form.Control id="messageData" className="mb-2" placeholder="Send Message" />
            </Col>
            <Col >
                <Button className="mb-2" style={{ float:"right" }} onClick={ () => {
                    sendMessage( document.getElementById( 'messageData' ).value );
                    document.getElementById( 'messageData' ).value = "";
                }}>
                    Send
                </Button>
            </Col>
        </Form.Row>
    </Form>
</div>
</div>
</Row>
</Container>
);
}

```

Note: Write down the contract address obtained from *Implementing the smart contract* section in the variable called `CONTRACT_ADDRESS` on line 9 of `App.jsx`.

Open the `index.js` file inside the `src` directory and paste the following code :

```

import React from "react";
import ReactDOM from "react-dom";
import { App } from "../src/App.jsx";

ReactDOM.render(
    <App /> ,
    document.getElementById('root')
);

```

Create a new file called `abi.js` in the `src` directory, and paste the following code :

```

export const abi = [
{
    "inputs": [
        {
            "internalType": "address",
            "name": "friend_key",
            "type": "address"
        },
        {
            "internalType": "string",
            "name": "name",
            "type": "string"
        }
    ],
    "name": "addFriend",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
},
{
    "inputs": [
        {
            "internalType": "address",
            "name": "pubkey",
            "type": "address"
        }
    ],
    "name": "checkUserExists",
    "outputs": [
        {
            "internalType": "bool",
            "name": "",
            "type": "bool"
        }
    ]
}
]

```

```

        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [
        {
            "internalType": "string",
            "name": "name",
            "type": "string"
        }
    ],
    "name": "createAccount",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
},
{
    "inputs": [],
    "name": "getMyFriendList",
    "outputs": [
        {
            "components": [
                {
                    "internalType": "address",
                    "name": "pubkey",
                    "type": "address"
                },
                {
                    "internalType": "string",
                    "name": "name",
                    "type": "string"
                }
            ],
            "internalType": "struct Database.friend[]",
            "name": "",
            "type": "tuple[]"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [
        {
            "internalType": "address",
            "name": "pubkey",
            "type": "address"
        }
    ],
    "name": "getUsername",
    "outputs": [
        {
            "internalType": "string",
            "name": "",
            "type": "string"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [
        {
            "internalType": "address",
            "name": "friend_key",
            "type": "address"
        }
    ],
    "name": "readMessage",
    "outputs": [
        {
            "components": [
                {
                    "internalType": "address",
                    "name": "sender",
                    "type": "address"
                }
            ]
        }
    ]
}
```

```

        },
        {
            "internalType": "uint256",
            "name": "timestamp",
            "type": "uint256"
        },
        {
            "internalType": "string",
            "name": "msg",
            "type": "string"
        }
    ],
    "internalType": "struct Database.message[]",
    "name": "",
    "type": "tuple[]"
}
],
"stateMutability": "view",
"type": "function"
},
{
    "inputs": [
        {
            "internalType": "address",
            "name": "friend_key",
            "type": "address"
        },
        {
            "internalType": "string",
            "name": "_msg",
            "type": "string"
        }
    ],
    "name": "sendMessage",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}
]

```

An Application Binary Interface (ABI) is a JSON object which stores the metadata about the methods of a contract like data type of input parameters, return data type & property of the method like payable, view, pure etc. You can learn more about the ABI from the [solidity documentation](#)

Now its time to run our React app. Use the following command to start the React app.

```
npm start
```

Walkthrough

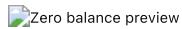
- Visit <http://localhost:3000> to interact with the app.
- User registration and adding a new friend
  Make sure your friend is also registered to the application while adding him as a friend.
- Chatting with friend
 

Conclusion

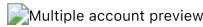
Congratulations! We have successfully developed a decentralized chat application which can be deployed on Avalanche. We also created a boilerplate React application to use as the frontend for our dapp. As a next step, You can improve the application by adding features like delete messages, block users, or create groups of friends. You can also optimize the gas cost by limiting the maximum number of messages stored.

Troubleshooting

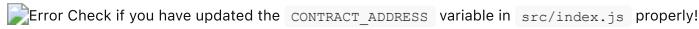
Transaction Failure

- Check if your account has sufficient balance at [Fuji block-explorer](#). You can find your address from the given [faucet](#)


- Make sure that you have selected the correct account on MetaMask if you have more than one account connected to the site.



Application Crash



About the Authors

The tutorial was created by [Nimish Agrawal](#) & [Sayan Kar](#). You can also reach out to them on LinkedIn [@Nimish Agrawal](#) and [@Sayan Kar](#).

References

- Smart contract deployment process - [Deploy a Smart Contract on Avalanche using Remix and MetaMask](#)

Distributed File Manager (DFM) using Avalanche, IPFS and ReactJS

Introduction

In this tutorial we will be making a **Distributed File Manager** using the **IPFS** protocol for storing our files, **Avalanche** network for storing the file references of each address to their uploaded files and **ReactJS** for the frontend code. For compiling and deploying our smart contracts, we will be using **Truffle Suite**.

For your information, [Truffle Suite](#) is a toolkit for launching decentralized applications dapps on the EVM. With Truffle you can write and compile smart contracts, build artifacts, run migrations and interact with deployed contracts. This tutorial illustrates how Truffle can be used with the [Avalanche](#) network, which is an instance of the EVM.

Prerequisites

- Basic familiarity with [Git](#), [NodeJS](#) and [npm](#).
- Basic familiarity with [ReactJS](#).
- Basic familiarity with [Avalanche](#) network, [Solidity](#) and [Truffle](#).

Requirements

- [NodeJS](#) >= 10.16 and [npm](#) >= 5.6 installed.
- [Truffle](#), which can be installed globally with `npm install -g truffle`
- [MetaMask](#) extension added to the browser.

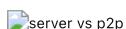
Understanding the Project

From the title, **Distributed File Manager**, you have got an idea that it's about making a dapp that will allow us to upload and manage files in a so-called **distributed** fashion. But you might be wondering, that what is **IPFS** and how will our files be distributed!!! Nothing to worry about, just go through the text, and all your doubts will be resolved.

Decoding IPFS and How is It Different?

IPFS is an acronym that stands for **InterPlanetary File System**. It is a communication protocol and network for storing and sharing data. Theoretically, it aims to make a file-sharing system that can communicate among the planets, someday. Check out [Awesome IPFS](#) to learn more about projects built on IPFS.

Currently, we are dominated by the **client-server** model of communication which is following **HTTP** aka **Hypertext Transfer Protocol**. This means that, in between the communication between two devices, one has to be the server (which will serve or respond with data) and the other should be the client (which will receive or request data). The major problem with this client-server model is that the client would have to request data from the server, far away from it, even if the same data was previously received by its neighbour or was available somewhere closer. This would cause high latency (delay in receiving data) and low bandwidths (speed of data transfer).



IPFS is a relatively new protocol, which aims to resolve these issues. It follows the **peer-to-peer** model of communication, in which there could be an arbitrary number of servers responding to the client with the required data. Once the client has the data (or even just bits of other data), it can then act as a server. Every node connected to the network can act as a server if it has the required software installed. Sending data from multiple servers may seem inefficient, however, the protocol is designed this way. The data is hashed and divided into pieces that can be transmitted and stored separately, but given sufficient, information can be re-joined later. Once all the pieces are in place, it makes the whole file.



IPFS is a large swarm of such nodes, which chose to serve data. We need IPFS clients to connect to those nodes and upload data. We can also connect to the network using the available JavaScript client libraries like `ipfs-http-client`. There are several providers like [Infura](#), which provides an HTTP portal to view the files on the IPFS. More technical details are provided ahead in the tutorial.

Initializing the Working Directory

Our application's client-side is made using **ReactJS**. Smart contracts will be made using the **Solidity** language and will be deployed on the **Avalanche** network with Truffle Suite. Therefore, we need to set up our working directory according to ReactJS and Truffle Suite, for making our development process smoother.

Open a terminal and navigate to the directory where we will create the application. Usually, this will be inside our user home directory but can be located wherever is practical. On most Linux distributions this will change into `/home/`. On macOS, it will be `/Users/`. On Windows, the user directories are located in `C:\Users`.

```
cd ~
```

Setting up the ReactJS Project

Create a new react app using `npx`. `npx` is an npm package runner (`x` stands for `eXecute`). The typical use is to download and run a package temporarily or for trials. You can learn more about `npx` [here](#).

```
npx create-react-app dfm-avalanche-react
```

Move to the newly created directory and install the basic dependencies.

```
cd dfm-avalanche-react
npm install --save dotenv web3 @truffle/contract @truffle/hdwallet-provider
```

Open the file `index.html` file inside of the `public` directory and replace the existing code with the following HTML :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Distributed File Manager</title>
    <link
      rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css"
    />
  </head>

  <body>
    <div id="root"></div>
  </body>
</html>
```

Open the file `App.js` inside of the `src` directory and replace the existing code with the following code:

```
import React from "react";

// 1. Importing other modules

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      web3: null,
      account: null,
      contract: null,
    };
  }

  componentDidMount() {
    this.init();
  }

  async init() {
    // 2. Load web3
    // 3. Load Account
    // 4. Load Smart-Contract instance
  }

  render() {
    return (
      <div>
        <font color="white">Distributed File Manager</font>
        {/* 5. Navbar */}

        {/* 6. IPFS Viewer component */}

        {/* 7. IPFS Uploader component */}
      </div>
    );
  }
}
```

```
}

export default App;
```

This `App` component will maintain a state with `web3` instance of the `MetaMask` provider for interacting with the Avalanche network, `account` address and instance of the deployed smart contract.

Open the file `index.js` inside of the `src` directory and replace the existing code with the following code:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
);
```

React project setup is now complete.

Setting up the Truffle Project

Run the following command in the root directory, to create a boilerplate for the `Truffle` project.

```
truffle init
```

This will set up Truffle's initial project structure. Smart contracts will be stored in the `contracts` folder, deployment functions for migrating smart contracts to the network will be stored in the `migrations` folder. By default, the `/build/contracts` folder would contain information about the compiled and deployed contract, ABI etc in the `.json` format and these files are known as `artifacts`.

There is also 1 **config** file created by the above command, which is, `truffle-config.js`. In this file, there is a lot of information regarding how to deploy contracts, how to choose a network to deploy them, and many others. Therefore, we should preserve this file for reference. So, use the below command to make a copy of this file. This would create a copy named `truffle-config-default.js`.

```
cp truffle-config.js truffle-config-default.js
```

Now we can update the `truffle-config.js` file with the following code, to deploy the smart contract on Avalanche's Fuji test network. This file is where we define the connection to the Avalanche network. An account on Avalanche with a valid mnemonic is required to deploy the contract to the network.

```
require("dotenv").config();
const HDWalletProvider = require("@truffle/hdwallet-provider");

// Account credentials from which our contract will be deployed
const MNEMONIC = process.env.MNEMONIC;

module.exports = {
  contracts_build_directory: "./src/build/contracts",
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*",
    },
    fuji: {
      provider: function () {
        return new HDWalletProvider({
          mnemonic: MNEMONIC,
          providerOrUrl: `https://api.avax-test.network/ext/bc/C/rpc`,
        });
      },
      network_id: "*",
      gas: 3000000,
      gasPrice: 470000000000,
      skipDryRun: true,
    },
  },
  compilers: {
    solc: {
      version: "0.8.0"
    }
  }
};
```

Note that we're setting the `gasPrice` and `gas` to the appropriate values for the Avalanche C-Chain. Here, you can see that we have used `contracts_build_directory` to change the default location of `artifacts` from the project root directory to the `src` folder. This is because React cannot access files that are present outside the `src` folder.

In the Truffle configuration file, you might have noticed a lot of jargon like HD wallet, mnemonic, address etc. Let's understand these keywords in more detail.

Before the concept of **wallets** or more specifically **Hierarchical Deterministic (HD) Wallets**, there were standalone Private-Public key pairs. HD wallets on the other hand are a tree of these key pairs, in which one key pair can generate multiple child key pairs.

In HD wallets, first, a random 128 - 256 bit number is generated, also known as **entropy** or the **root seed** or **private key**. This entropy is then appended with few bits of its checksum, to make the number of bits in entropy, a multiple of 11. Then this sequence of bits is divided into sections of 11 bits each. Each section represents the index of a word in a [list](#) of 2048 words. This sequence of words represents our wallet's **mnemonic**.

Checksum is a few bits of the hash of data to detect errors that may have been introduced during its transmission or storage. Errors are verified by hashing the actual data and comparing it with its appended checksum. If the checksum matches with the hashed data then there is no error.

Then the root seed is passed to a one-way hash function to generate a 512-bit seed. The left 256 bits will make the **master private key** and the right 256 bits will make the **master chain code**. Chain codes are used to introduce randomness in the child keys. HD wallets use the **child key derivation (CKD)** function to derive children keys from parent keys. The child key derivation functions are based on a one-way hash function that combines:

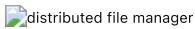
- A parent private or public key
- A seed called a chain code (256 bits)
- An index number (32 bits)



The index number can range from 0 to $2^{32} - 1$. Thus using a parent with a given private key and chain code we can generate 2^{32} or around 4 Billion child key pairs. In a normal derivation, we use parent public key and chain code to generate children. But this could be vulnerable to security threats and hence we can make derivation hard by using the parent's private key instead of the public key for CKD. This process is known as **Hardened child key derivation**. And to distinguish it from normal derivation, we use different index numbers. For normal derivation index number is from 0 to $2^{31} - 1$ and for hardened derivation, it is from 2^{31} to $2^{32} - 1$. Hardened index number start from 2 Billion which make it difficult to read, so we use i' to represent index $2^{31} + i$, where $0 \leq i \leq 2^{32} - 1$.



Master keys along with master chain code can create child keys which can further create grandchild keys and so on. Each generation is known as a tree level. Keys in an HD wallet are identified using a **path** naming convention, with each level of the tree separated by a slash (/) character. Private keys derived from the master private key start with **m**. Public keys derived from the master public key start with **M**. An HD path `m/0` represents the 0th or first child private key derived from the master. Similarly, `m/3/1` denotes the 2nd child private key of the 4th or ($2^{31} + 3$)th hardened child derived from the master.



There are various Bitcoin Improvement Proposals (BIP) that proposes the standard way of deriving paths. BIP0044 (44th proposal) specifies the structure as consisting of five predefined tree levels:

```
m / purpose' / coin_type' / account' / change / address_index
```

- **purpose** - Always set to 44'.
- **coin_type** - Specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its subtree under the second level.
- **account** - Allows users to subdivide their wallets into separate logical subaccounts, for accounting or organizational purposes.
- **change** - It has 2 subtrees, one normal receiving address and the other for receiving change tokens which are reverted when you supplied more than the required transaction cost.
- **address_index** - We can use all the 4 Billion child keys as our address, but this index would set the primary address for our wallet.

Avalanche's wallets like <https://wallet.avax.network> use the path `m/44'/9000'/0'/0` for its key derivation, since the coin type of Avalanche is **52752**. The list of different crypto coins along with their type can be found [here](#). The coin types have nothing to do with the blockchain architecture and is chosen randomly. By default `ethers.Wallet.fromMnemonic()` function uses Ethereum's default path which has a coin type of **60**. That's why we need to manually set the path in this function. Using a different path would give a different address derived from the same mnemonic. So, if we want to use our manual paths, we should remember them, otherwise, we can't derive the address without a path.

I would recommend you to read more about these keys, addresses and wallets on [O'Reilly](#).

Get Avalanche's Credentials

For deploying smart contracts we need two things: A node connected to the Avalanche network and an account with few AVAX. Avalanche connected node through RPC (Remote Procedure Call) is provided for free by the Avalanche Networks.

Now we need an Avalanche wallet, where we would keep our funds, required for all the transactions on the network. So, visit [here](#) and create an account. Save the mnemonic in a secure place (we would need it later). Instructions to add funds will be provided later in the tutorial.

Add .env File

Now we need a **Avalanche** wallet, where we would keep our funds, required for all the transactions on the network. Visit the [Avalanche Wallet](#) and create an account. While creating your account you will see your public address and mnemonic. This public address will be required to transfer funds. Save the **mnemonic** in a secure place, we will need it later. Instructions to add funds will be provided later in the tutorial.

Create a `.env` file in the project root folder. Please take a note that dot (.) is necessary for the `.env` file name. Now copy your Avalanche wallet's mnemonic in the `.env` file as shown below. In the `.env` file, **MNEMONIC** should be enclosed within double-quotes ("").

```
MNEMONIC=""
```

Never share or commit your `.env` file. It contains your credentials like `mnemonics`. Therefore, it is advised to add `.env` to your `.gitignore` file.

Our project setup is now complete.

To confirm that we are on the same page, run the following command in the project root folder.

```
npm start
```

It might take few seconds, to show output as in the image below.



In a web browser, visit the URL <http://localhost:3000>. If npm start has not encountered any errors, we will see the text "Distributed File Manager" at the top of the page as shown in this image :



Create the FileManager Contract

Create the file `FileManager.sol` (`.sol` stands for Solidity) inside of the `contracts` directory and paste the following code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.0;

contract FileManager {
    // Structure of each File
    struct File {
        string fileName;
        string fileType;
        string cid;
    }

    // Mapping of each user's address with the array of files they are storing
    mapping(address => File[]) files;

    function addFile(string[] memory _fileInfo, string memory _cid) public {
        files[msg.sender].push(File(_fileInfo[0], _fileInfo[1], _cid));
    }

    function getFiles(address _account) public view returns (File[] memory) {
        return files[_account];
    }
}
```

`FileManager` is a solidity smart contract that lets us store and view the meta details of file which we upload on the IPFS network. IPFS uses content addressing rather than location addressing. To identify files on the network, IPFS uses a cryptographic hash for each file. This hash is known as a content identifier or cid. Whatever the size of the file, the length of this hash would be the same and is enough to identify every file uniquely. After uploading the file to IPFS, we get a cid that acts as a reference to that file on the network, and we store this cid on the Avalanche blockchain.

Now IPFS generates a unique content identifier, the "cid" which can be used to reference this file on the network. cid's look similar to `QmVwyUH96NeQPwLN5jDkgNxM41xGCB1EVjnBYX7NoWWmKH`, a long string of upper- and lower-case letters and numbers. This is the hashed representation of the content. More information about IPFS content identifiers can be found on the [IPFS docs](#).

Let's understand this smart contract

The code for smart contract is everything within `contract FileManager { }`.

Basic structure about Files - `File` is a struct that's a skeleton to store the details of each file. We are having three attributes of each file: `fileName`, `fileType` that is whether it is image, audio, video or an application and finally `cid`. Here, `files` is a mapping between the owner (address) of the files and the array of those `File` structures which they uploaded.

```
// Structure of each File
struct File {
    string fileName;
    string fileType;
    string cid;
}

// Mapping of each user's address with the array of files they are storing
mapping(address => File[]) files;
```

Adding files - `addFile()` function is used to add details of file to the array of `File` structures corresponding to each address. `files[msg.sender]` refers to the array of file structures, belonging to the caller of this function, that is the address `msg.sender`. Function's arguments are `_fileInfo[]` which is an

array of 2 parameters (file name and file type respectively) and the second argument is `cid` which is the content id for the uploaded file.

```
function addFile(string[] memory _ fileInfo, string memory _cid) public {
    files[msg.sender].push(File(_ fileInfo[0], _ fileInfo[1], _cid));
}
```

Viewing stored files - `getFiles()` is a function that returns the array of file structures corresponding to the account address. It returns the details of all the files as an array that's being uploaded by the address (passed in the argument of this function) on the IPFS network.

```
function getFiles(address _account) public view returns (File[] memory) {
    return files[_account];
}
```

Make a New File for Migrating Smart Contracts

Create a new file in the `migrations` directory named `2_deploy_contracts.js`, and add the following block of code. This handles deploying the `FileManager` smart contract to the blockchain.

```
const FileManager = artifacts.require("FileManager");

module.exports = function (deployer) {
    deployer.deploy(FileManager);
};
```

Compile Contracts with Truffle

If we have altered the code within our Solidity source files or made new ones (like `FileManager.sol`), we need to run `truffle compile` in the terminal, from inside the project root directory.

```
truffle compile
```

You should see:

```
Compiling your contracts...
=====
> Compiling ./contracts/FileManager.sol
> Artifacts written to /home/guest/blockchain/dfm-avalanche-react/src/build/contracts
> Compiled successfully using:
  - solc: 0.8.0+commit.c7dfd78e.Emscripten clang
```

There might be an error Error: Cannot find module 'pify', if the pify module is not installed automatically while installing truffle. So, this issue can be resolved by separately installing pify, using the command below

```
npm install pify --save
```

Compiling the smart contracts would create `.json` file in the `src/build/contracts` directory. It stores `ABI` and other necessary metadata.

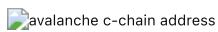
`ABI` refers to Application Binary Interface, which is a standard for interacting with the smart contracts from outside the blockchain as well as contract-to-contract interaction. Please refer to the Solidity's documentation about ABIs [here](#) to learn more.

Fund the Account and Run Migrations on Avalanche's Fuji Test Network.

When deploying smart contracts to the Avalanche network, it will require some deployment cost. As you can see inside `truffle-config.js`, `@truffle/hdwallet-provider` will help us in deploying on Avalanche and deployment cost will be managed by an account whose mnemonic has been stored in the `.env` file. Therefore we need to fund the account.

Fund Your Account

We need funds in our C-Chain address, as smart contracts are deployed on C-Chain, the Contract-Chain. This address can easily be found on the [Avalanche Wallet](#) dashboard. Avalanche network has 3 chains: X-Chain, P-Chain and C-Chain. The address of all these chains can be found by switching tabs at the bottom of the division, where there is a QR code. So, switch to C-Chain, and copy the address. Now fund your account using the faucet link [here](#) and paste your C-Chain address in the input field. Refer to the below image, to identify the address section.



You'll need to send at least 135422040 nAVAX to the account to cover the cost of contract deployments. Here nAVAX refers nano-AVAX, that is a billionth of an AVAX, or simply 1 nAVAX = (1/1000,000,000) AVAX. Though funding through faucet would give you enough AVAX to run multiple deployments and transactions on the network.

Run Migrations

Now everything is in place to run migrations and deploy the `FileManager`:

```
truffle migrate --network fuji
```

This might take a while depending upon your internet connection or traffic on the Avalanche network.

Note - For development purpose, we may deploy our contracts on a local network, by running Ganache (Truffle's local blockchain simulation) and using the command

```
truffle migrate --network development
```

On successful execution of this command, you should see:

```
Starting migrations...
=====
> Network name:      'fuji'
> Network id:        1
> Block gas limit:   8000000 (0x7a1200)

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash:  0x094a9c0f12ff3158bcb40e266859cb4f34a274ea492707f673b93790af40e9e9
> Blocks: 0          Seconds: 0
> contract address:  0x0bf00d0Af6d5c864f86E6b96216e0a2Da111055
> block number:      40
> block timestamp:   1620393171
> account:           0x80599dd7F8c5426096FD189dccC6C40f47e8e3714
> balance:            39.71499696
> gas used:          173118 (0x2a43e)
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.00346236 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:         0.00346236 ETH

2_deploy_contracts.js
=====

Deploying 'FileManager'
-----
> transaction hash:  0xebef13fc6bbe250eea9151faf02bfe247ec497294acc84c9b8319ed609ced086
> Blocks: 0          Seconds: 0
> contract address:  0xf30D372A6911CCF6BBale84c3CEd51cC0F3D7769
> block number:      42
> block timestamp:   1620393172
> account:           0x80599dd7F8c5426096FD189dccC6C40f47e8e3714
> balance:            39.69235442
> gas used:          1090212 (0x10a2a4)
> gas price:          20 gwei
> value sent:         0 ETH
> total cost:         0.02180424 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost:         0.02180424 ETH

Summary
=====
> Total deployments:  2
> Final cost:         0.0252666 ETH
```

Deploying smart contracts is a transaction on the blockchain network. Therefore, in the above output, you can see a **transaction hash** which starts with a `0x`. Using this transaction hash you can verify your transaction on the Avalanche's Fuji network, using their blockchain explorer [here](#) by searching your transaction hash.

You may also view your deployed smart contracts using their contract address as provided in the above output of `truffle migrate`.

Possible Errors and Troubleshooting

If you didn't create an account on the C-Chain you'll see this error:

```
Error: Expected parameter 'from' not passed to function.
```

If you didn't fund the account, you'll see this error:

```
Error: *** Deployment Failed ***

"Migrations" could not deploy due to insufficient funds
* Account: 0x090172CD36e9f4906Af17B2C36D662E69f162282
* Balance: 0 wei
* Message: sender doesn't have enough funds to send tx. The upfront cost is: 1410000000000000000000000 and the sender's account
only has: 0
* Try:
+ Using an adequately funded account
```

The information like contract address and ABI of the deployed contract is present in the `src/build/contract` directory as `FileManager.json`.

Building the User Interface

We have already set up our React project directory. The client-side files to interact with the Avalanche blockchain are present in the `src` folder. First, we will make a ReactJS component with a couple of functions to connect our browser with the Avalanche network. These functions will be kept in a separate file named `BlockchainUtil.js`.

BlockchainUtils Component

Create the file `BlockchainUtil.js` inside of the project `src` directory and paste the following code:

```
import React from "react";
import Web3 from "web3";
import TruffleContract from "@truffle/contract";

// For connecting our web application with MetaMask Web3 Provider
export class GetWeb3 extends React.Component {
  async getWeb3() {
    let web3 = window.web3;
    if (typeof web3 !== "undefined") {
      // Setup Web3 Provider
      this.web3Provider = web3.currentProvider;
      this.web3 = new Web3(web3.currentProvider);
      return this.web3;
    } else {
      this.isWeb3 = false;
    }
  }
}

// For getting our Smart-Contract's instance to interact with it using javascript
export class GetContract extends React.Component {
  async getContract(web3, contractJson) {
    // Setup Contract
    this.contract = await TruffleContract(contractJson);
    this.contract.setProvider(web3.currentProvider);
    return await this.contract.deployed();
  }
}

// For getting our account address from the MetaMask
export class GetAccount extends React.Component {
  async getAccount(web3) {
    return await web3.eth.getAccounts();
  }
}
```

Updating App.js - `App.js` is the entry point of any React application. Therefore we need to update `App.js` regularly with the components which we want to show in our application. As we move further, build all components, we will also update `App.js` in the end.

IPFSUploader Component

Now let's make a component that will upload the files from our system to the IPFS network. So, make a file named `IPFSUploader.js` in the `src` directory and use the code as present in this [file](#).

Let's understand this component block by block.

IPFS Client - First we need to make a connection to IPFS client using the `ipfs-http-client` module. This has to be done by some IPFS provider like `Infura`. So, the following line would create an IPFS client -

```
const ipfs = create({ host: 'ipfs.infura.io', port: 5001, protocol: 'https' })
```

state - `IPFSUploader` component will maintain a state of file properties like `fileName`, `fileType`, `buffer` of each file, account address and `cid` of the uploaded file. These state variables will be updated whenever there is a change in the input field of file type.

captureFile() - This function will be called whenever there is an `onChange` event in the input field. This will update the state with necessary file information. In this function, we will be having a `Compressor` instance which will compress the file of `image` type.

onSubmit() - This function would be called when the user will submit the form containing the file as an input. This function would first invoke the `add()` function of the IPFS client and upload the `buffer` of this file which was previously stored in the `state` of this component. Once the file is uploaded, it will return a `cid`. After that, we will add this file information along with `cid` to the smart contract using the `addFile()` contract function.

Since we have used new libraries like `compressorjs`, `ipfs-http-client` and `rimble-ui` which we did not install previously, therefore please install these libraries using the below command.

```
npm install --save ipfs-http-client compressorjs rimble-ui --force
```

Rimble UI library comes with a peer dependency of `react@16.9.0` which is not the latest version of React that we are using which is `react@17.0.2`. Running the `npm install` command without the `--force` tag would cause an unable to resolve dependency tree conflict. Thus, the `--force` tag is used to override any type of conflicts and proceeds with the installation anyway. Another way to resolve this conflict is by using the `--legacy-peer-deps` tag instead, but this would ignore all peer dependencies which we do not require, as the conflict is only between `react` and `rimble-ui`.

IPFSViewer Component

Now make a new file named `IPFSViewer.js`. This component would be used to fetch file information from the deployed smart contract and display it on the website. Use the code as present in this [file](#).

Let's understand the above component block by block.

state - The state of this component would contain an array of different types of files like `imageFiles`, `audioFiles` etc.

loadFiles() - This function would be called after the component is mounted and would load the state with all the files which are uploaded by the account address. Using the `getFiles()` function of the smart contract, it can easily fetch all file information from the blockchain. It will separate the different types of files like images, videos, audios etc. accordingly and update the state.

showImageFiles() - This function would return components with all image files composed with proper `img` tag, as an array, to the caller of this function.

Similarly, there are different functions for each file type.

IPFSViewer Stylesheet

`IPFSViewerCSS.css` file has been imported to add few designs to the page like decreasing the width of the scroll bar, colour changes etc. So, make a new file named `IPFSViewerCSS.css` and add the following code inside it.

```
.imageViewer::-webkit-scrollbar {  
    height: 2px;  
}  
  
.imageViewer::-webkit-scrollbar-track {  
    box-shadow: inset 0 0 5px rgb(94, 94, 94);  
    border-radius: 50px;  
}  
  
.imageViewer::-webkit-scrollbar-thumb {  
    background: rgb(0, 0, 0);  
    border-radius: 50px;  
}  
  
a {  
    color: white;  
}
```

Now we need to update our `App.js` file with all the components that we have made so far.

Import Modules - First import all the modules and components into the `App.js` file by appending the following code under the `//1 Importing...` section.

```
// 1. Importing other modules  
import {GetWeb3, GetContract, GetAccount} from './BlockchainUtil';  
import IPFSUploader from './IPFSUploader';  
import IPFSViewer from './IPFSViewer';  
  
import contractJson from './build/contracts/FileManager.json';
```

Load Web3 - Now put the following code under the `//2. Load web3...` section. This would set the state with web3 instance.

```
// 2. Load web3
const Web3 = new GetWeb3();
this.web3 = await Web3.getWeb3();
this.setState({web3: this.web3});
```

Load Account - Put the following code under the `//3. Load Account...` section. This would set the state with MetaMask wallet's first connected address.

```
// 3. Load Account
const Account = new GetAccount();
this.account = await Account.getAccount(this.web3);
this.setState({account: this.account[0]});
```

Load Smart contract - Put the following code under the `//4. Load smart...` section. This would set the state with deployed smart contract's instance for the contract's interaction using JavaScript.

```
// 4. Load Contract
const Contract = new GetContract();
this.contract = await Contract.getContract(this.web3, contractJson);
this.setState({contract: this.contract});
```

Load components - Inside the `<div>` tag of `return()` function, replace the existing sample text `Distributed File Manager` with the code of the following components.

```
/* 5. Navbar */
<nav className="navbar navbar-dark shadow" style={{backgroundColor: "#1b2021", height: "60px", color: "white"}}>
  <b>Distributed File Manager</b>
  <span style={{float: "right"}}>{this.state.account}</span>
</nav>

/* 6. IPFS Viewer component */
<IPFSViewer state = {this.state} />

/* 7. IPFS Uploader component */
<IPFSUploader state = {this.state} />
```

Your `App.js` would now look like this [file](#)

Starting the Application

Now go to the project root directory of the project, the `dfm-avalanche-react` directory, and run the command `npm start`. The ReactJS server would start automatically. Visit <http://localhost:3000> to interact with the built dapp.

Don't forget to set up MetaMask with Avalanche Fuji testnet and also fund the account with Avalanche test tokens to upload files.

In the MetaMask extension, add a custom RPC by clicking at the network dropdown in the centre of the extension. Fill in the details as shown in the below image.

Info	Value
Network Name	Avalanche Fuji
New RPC URL	https://api.avax-test.network/ext/bc/C/rpc
Chain ID	43113
Currency Symbol	AVAX-C
Block Explorer URL	https://testnet.snowtrace.io

If you find any difficulty in setting up the project, then feel free to clone this repository <https://github.com/rajanjan0608/dfm/tree/avalanche>, and follow the steps in the `README.md` file of this repo in order to run the application.



Conclusion

Congratulations! You have successfully built a Distributed File Manager by deploying the smart contract on Avalanche Fuji test network using **Truffle Suite** and connecting your client-side made with **ReactJS**. The most interesting part is that we have used the **IPFS** protocol for uploading our files on the distributed network.

What's Next?

We have built a Distributed File Manager with basic upload and view features. I want to encourage you to make a more scalable and sophisticated application by adding few more features like encrypting files before uploading by using the receiver's public key or their account address. Or if you want to keep the file to yourself, you might encrypt the file using your account address. Learn more on how to encrypt files before uploading them to IPFS [here](#).

About the Author

This tutorial is created by [Raj Ranjan](#).

App.js

```
import React from 'react';
import {GetWeb3, GetContract, GetAccount} from './BlockchainUtil';
import IPFSUploader from './IPFSUploader';
import IPFSViewer from './IPFSViewer';

import contractJson from './build/contracts/FileManager.json';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      web3: null,
      account: null,
      contract: null
    }
  }

  componentDidMount() {
    this.init();
  }

  async init() {
    //1. Load web3
    const Web3 = new GetWeb3();
    this.web3 = await Web3.getWeb3();
    this.setState({web3: this.web3});

    //2. Load Account
    const Account = new GetAccount();
    this.account = await Account.getAccount(this.web3);
    this.setState({account: this.account[0]});

    //3. Load Contract
    const Contract = new GetContract();
    this.contract = await Contract.getContract(this.web3, contractJson);
    this.setState({contract: this.contract});

    console.log("App contract: ", this.contract);
  }

  render() {
    return (
      <div>
        /* 5. Navbar */
        <nav className="navbar navbar-dark shadow" style={{backgroundColor: "#1b2021", height: "60px", color: "white"}}>
          <b>Distributed File Manager</b>
          <span style={{float: "right"}}>{this.state.account}</span>
        </nav>

        /* 6. IPFS Viewer component */
        <IPFSViewer state = {this.state} />

        /* 7. IPFS Uploader component */
        <IPFSUploader state = {this.state} />
      </div>
    )
  }
}

export default App;
```

IPFSUploader.js

```
import React from "react";
import Compressor from "compressorjs";
import { Loader } from "rimble-ui";
```

```

const { create } = require("ipfs-http-client");
const ipfs = create({ host: "ipfs.infura.io", port: 5001, protocol: "https" });

class IPFSUploader extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      buffer: null,
      fileName: null,
      fileType: null,
      cid: null,
      account: this.props.state.account,
      loading: false,
      loadingReason: "",
    };
  }

  captureFile = (event) => {
    event.preventDefault();
    const file = event.target.files[0];
    let type = file.type.split("/");

    if (type[0] === "image") {
      new Compressor(file, {
        quality: 0.2,
        success: (compressedResult) => {
          const reader = new window.FileReader();
          reader.readAsArrayBuffer(compressedResult);
          reader.onloadend = () => {
            this.setState({
              buffer: Buffer(reader.result),
              fileName: file.name,
              fileType: file.type,
            });
          };
        },
      ));
    } else {
      const reader = new window.FileReader();
      reader.readAsArrayBuffer(file);
      reader.onloadend = () => {
        this.setState({
          buffer: Buffer(reader.result),
          fileName: file.name,
          fileType: file.type,
        });
      };
    }
  };
}

onSubmit = async (event) => {
  event.preventDefault();

  this.setState({ loading: true, loadingReason: "Uploading to IPFS" });
  const { cid } = await ipfs.add(this.state.buffer);
  this.setState({ cid: cid.string, loadingReason: "Waiting for approval" });

  try {
    await this.props.state.contract.addFile(
      [this.state.fileName, this.state.fileType],
      this.state.cid,
      { from: this.props.state.account, gas: 20000000 }
    );
  } catch (e) {
    console.log("Transaction failed");
  }

  this.setState({ loading: false });
};

render() {
  return (
    <div>
      <center style={{ margin: "50px auto" }}>
        <form onSubmit={this.onSubmit}>
          <input
            className="text-light bg-dark"
            type="file"

```

```

        onChange={this.captureFile}
    />
<br />
<br />

{this.state.loading ? (
    <center>
        <Loader size="40px" color="white" />
        <br />
        <font size="2" color="white" style={{ marginTop: "-10px" }}>
            {this.state.loadingReason}
        </font>
    </center>
) : (
    <input className="btn btn-dark" type="submit" />
)
</form>
</center>
</div>
);
}

export default IPFSUploader;

```

IPFSViewer.js

```

import React from "react";
import App from "./app";
import IPFSViewerCSS from "./IPFSViewerCSS.css";

class IPFSViewer extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            imageFiles: [],
            videoFiles: [],
            applicationFiles: [],
            audioFiles: [],
            otherFiles: []
        };
    }

    app = null;

    async componentDidMount() {
        this.app = new App();
        await this.app.init();
        await this.loadFiles();
    }

    loadFiles = async () => {
        const files = await this.app.contract.getFiles(this.app.account[0]);
        let imageFiles = [], videoFiles = [], audioFiles = [], applicationFiles = [], otherFiles = [];

        files.forEach((file) => {
            let type = file[2].split("/");
            if (type[0] === "image") {
                imageFiles.push(file);
            } else if (type[0] === "video") {
                videoFiles.push(file);
            } else if (type[0] === "audio") {
                audioFiles.push(file);
            } else if (type[0] === "application") {
                applicationFiles.push(file);
            } else {
                otherFiles.push(file);
            }
        });

        this.setState({
            imageFiles,
            videoFiles,
            audioFiles,
            applicationFiles,
            otherFiles
        });
    };

    this.setState({
        imageFiles,
        videoFiles,
        audioFiles,
        applicationFiles,
        otherFiles
    });
}

```

```

        otherFiles,
    );
};

showImageFiles = () => {
    let fileComponent = [];

    this.state.imageFiles.forEach((file) => {
        let fileName;
        if (file[1].length < 12) {
            fileName = file[1];
        } else {
            fileName = file[1].substring(0, 6) + "..." + file[1].substring(file[1].length - 8, file[1].length);
        }

        fileComponent.push(
            <a href={`https://ipfs.infura.io/ipfs/${file[3]}`}>
                <img
                    alt={fileName}
                    src={`https://ipfs.infura.io/ipfs/${file[3]}`}
                    style={{
                        margin: "5px",
                        width: "200px",
                        height: "150px",
                        border: "solid white 2px",
                        borderRadius: "5px",
                    }}
                />
                <center>{fileName}</center>
            </a>
        );
    });
    return fileComponent;
};

showVideoFiles = () => {
    let fileComponent = [];
    this.state.videoFiles.forEach((file) => {
        let fileName;
        if (file[1].length < 12) {
            fileName = file[1];
        } else {
            fileName = file[1].substring(0, 6) + "..." + file[1].substring(file[1].length - 8, file[1].length);
        }

        fileComponent.push(
            <div>
                <video
                    src={`https://ipfs.infura.io/ipfs/${file[3]}#t=0.1`}
                    controls
                    style={{
                        margin: "5px",
                        width: "290px",
                        height: "200px",
                        border: "solid white 2px",
                        borderRadius: "5px",
                    }}
                />
                <center>{fileName}</center>
            </div>
        );
    });
    return fileComponent;
};

showAudioFiles = () => {
    let fileComponent = [];
    this.state.audioFiles.forEach((file) => {
        let fileName;
        if (file[1].length < 12) {
            fileName = file[1];
        } else {
            fileName = file[1].substring(0, 6) + "..." + file[1].substring(file[1].length - 8, file[1].length);
        }

        fileComponent.push(
            <div>
                <audio
                    src={`https://ipfs.infura.io/ipfs/${file[3]}#t=0.1`}
                    controls

```

```

        style={{ margin: "10px" }}
      />
      <center>{fileName}</center>
    </div>
  );
);
return fileComponent;
};

showApplicationFiles = () => {
let fileComponent = [];
this.state.applicationFiles.forEach((file) => {
  let fileName;
  if (file[1].length < 12) {
    fileName = file[1];
  } else {
    fileName = file[1].substring(0, 6) + "..." + file[1].substring(file[1].length - 8, file[1].length);
  }
  fileComponent.push(
    <div style={{ width: "120px" }}>
      <center
        style={{ cursor: "pointer" }}
        onClick={() => {
          window.location.href = `https://ipfs.infura.io/ipfs/${file[3]}`;
        }}
      >
        <a href={`https://ipfs.infura.io/ipfs/${file[3]}`}>
          <img
            alt={fileName}
            src={
              "https://img2.pngio.com/filetype-docs-icon-material-iconset-zhoologo-file-icon-png-256_256.png"
            }
            style={{ width: "50px", height: "50px" }}
          />
          <br />
          {fileName}
        </a>
      </center>
    </div>
  );
));
return fileComponent;
};

showOtherFiles = () => {
let fileComponent = [];
this.state.otherFiles.forEach((file) => {
  let fileName;
  if (file[1].length < 12) {
    fileName = file[1];
  } else {
    fileName =
      file[1].substring(0, 6) +
      "..." +
      file[1].substring(file[1].length - 8, file[1].length);
  }
  fileComponent.push(
    <div style={{ width: "120px" }}>
      <center style={{ cursor: "pointer" }}>
        <a href={`https://ipfs.infura.io/ipfs/${file[3]}`}>
          <img
            alt={fileName}
            src={
              "https://images.vexels.com/media/users/3/152864/isolated/preview/2e095de08301a57890aad6898ad8ba4c-yellow-circle-question-mark-icon-by-vexels.png"
            }
            style={{ width: "50px", height: "50px" }}
          />
          <br />
          {fileName}
        </a>
      </center>
    </div>
  );
));
return fileComponent;
};

```

```

    render() {
      let imageFiles = this.showImageFiles(), videoFiles = this.showVideoFiles(), audioFiles = this.showAudioFiles(),
      applicationFiles = this.showApplicationFiles(), otherFiles = this.showOtherFiles();

      return (
        <div style={{ margin: "20px" }}>
          <b style={{ color: "white" }}>Images</b> <br /><br />

          <div
            className="imageViewer"
            style={{
              color: "white",
              height: "200px",
              display: "flex",
              overflowX: "scroll",
            }}
          >
            {imageFiles.length === 0 ? "No files to show" : imageFiles}
          </div> <br /><br />

          <b style={{ color: "white" }}>Videos</b> <br /><br />
          <div
            className="imageViewer"
            style={{
              color: "white",
              height: "250px",
              display: "flex",
              overflowX: "scroll",
            }}
          >
            {videoFiles.length === 0 ? "No files to show" : videoFiles}
          </div> <br /><br />

          <b style={{ color: "white" }}>Audio</b> <br /><br />

          <div
            className="imageViewer"
            style={{
              color: "white",
              height: "250px",
              display: "flex",
              overflowX: "scroll",
            }}
          >
            {audioFiles.length === 0 ? "No files to show" : audioFiles}
          </div> <br /><br />

          <b style={{ color: "white" }}>Applications</b> <br /><br />

          <div
            className="imageViewer"
            style={{
              color: "white",
              height: "150px",
              display: "flex",
              overflowX: "scroll",
            }}
          >
            {applicationFiles.length === 0 ? "No files to show" : applicationFiles}
          </div> <br /><br />

          <b style={{ color: "white" }}>Others</b> <br /><br />

          <div
            className="imageViewer"
            style={{
              color: "white",
              height: "150px",
              display: "flex",
              overflowX: "scroll",
            }}
          >
            {otherFiles.length === 0 ? "No files to show" : otherFiles}
          </div>
        </div>
      );
    }
  }
}

```

```
export default IPFSViewer;
```

Create an Auction Bidding on Avalanche using ReactJS

Learn how to create an Avalanche-based auction bidding with a ReactJS frontend.

Introduction

We will learn how to build smart contracts by making an auction bidding and deploy them on Avalanche and interact with them using ReactJS and Drizzle.

We are going to generate [ReactJS](#) boilerplate code using `create-react-app`, which we will modify for our auction bidding frontend. React is useful for the frontend due to its efficiency and user-friendly blockchain interaction. For the backend, [Solidity](#) smart contracts will be deployed to the Avalanche blockchain using [Truffle Suite](#).

Prerequisites

- Basic familiarity with [NodeJS](#) and [npm](#).
- Basic familiarity with [ReactJS](#), [React context APIs](#) and [Drizzle](#).
- Basic familiarity with [Avalanche](#) network, [Solidity](#) and [Truffle](#).

Requirements

- [NodeJS](#) >= 10.16 and [npm](#) >= 5.6 installed.
- [Truffle](#), which can be installed globally with `npm install -g truffle`
- [MetaMask](#) extension added to the browser.

Note: Do not download MetaMask from an unofficial source.

Initialize the Project Directory & Install All Dependencies

The developer needs to set up a working directory according to ReactJS and Truffle.

Follow the steps below to create the application. Open up the terminal and navigate to the directory where you would like to create this application. Now create a new directory with `mkdir <directory_name>` command. Change the current directory to this newly created directory using `cd <directory_name>`. For instance: if we name it `bid`, then

```
mkdir bid  
cd bid
```

Setup the ReactJS Project

Let us now create a new react app using `npx` (`npm` package runner). The typical use is to download and run a package temporarily or for trials. Using `npx` to execute the package binaries for `create-react-app` will generate a new React app scaffold in the specified directory.

```
npx create-react-app client
```

Now change to the recent directory "client" using `cd client` command and install the required dependencies using

```
npm install --save dotenv web3 @truffle/contract @truffle/hdwallet-provider @drizzle/store
```

Open the file `index.js` inside the `src` directory and replace the existing code with the following code.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

import { Drizzle } from '@drizzle/store';
import drizzleOptions from './drizzleOptions';
import { DrizzleProvider } from './drizzleContext';

const drizzle = new Drizzle(drizzleOptions);

ReactDOM.render(
  <DrizzleProvider drizzle={drizzle}>
    <App/>
  </DrizzleProvider>
,
  document.getElementById('root')
);
```

Next step: Open up the `App.js` file present inside the `src` directory and replace the existing code with the following code mentioned below.

```

import Auction from "./Auction";
import { useDrizzleContext } from "./drizzleContext";

function App() {
  const {drizzleVariables} = useDrizzleContext();

  if(!drizzleVariables.initialized) {
    return "Loading..."
  } else {
    return <Auction/>
  }
}

export default App;

```

By doing the above steps, the ReactJS project setup is completed.

Setup the Truffle Project

To create a boilerplate for the Truffle project execute the following command in the project root directory.

```
truffle init
```

Now, the initial project is being set up. Solidity code will be stored in the `contracts` directory. Deployment functions written in JavaScript will be stored in the `migrations` folder. By default, the `/build/contracts` folder contains information about the compiled and deployed contract, like the ABI, in JSON format. These meta-files are commonly referred to as `artifacts`.

`truffle-config.js` is another `config` file created by the `truffle init` command. This file carries a lot of information like how to deploy a contract, how to choose a network to deploy the following contract, and much more. Thus it became a priority to save this file for reference and we can create a copy of this file using the command mentioned below.

```
cp truffle-config.js truffle-config-default.js
```

Now, the name of the copied file will be `truffle-config-default.js`. Consequently, we will update the `truffle-config.js` file, with the information needed to deploy the smart contract on the Fuji test network. This file helps us in connecting to the Avalanche node, and we will require an Avalanche wallet mnemonic for deploying the contract on the network.

```

require("dotenv").config();
const HDWalletProvider = require("@truffle/hdwallet-provider");

// Account credentials from which our contract will be deployed
const MNEMONIC = process.env.MNEMONIC;

module.exports = {
  contracts_build_directory: "./src/build/contracts",
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*",
    },
    fuji: {
      provider: function () {
        return new HDWalletProvider({
          mnemonic: MNEMONIC,
          providerOrUrl: `https://api.avax-test.network/ext/bc/C/rpc`,
        });
      },
      network_id: "*",
      gas: 3000000,
      gasPrice: 470000000000,
      skipDryRun: true,
    },
  },
  compilers: {
    solc: {
      version: "0.8.0"
    }
  }
};

```

Here we are setting the `gas` and `gasprice` to an appropriate value of the Avalanche C-chain. The developer may observe that `contract_build_directory` is being used to change the default location of `artifacts` from the project root directory to the `src` folder. Since React is unable to access the files present outside the `src` folder.

Receive Avalanche Credentials

For deploying smart contracts we need two things: A node connected to the Avalanche network and an account with few AVAX. Avalanche connected node through RPC (Remote Procedure Call) is provided for free by the Avalanche Networks.

Now we need an Avalanche wallet, where we would keep our funds, required for all the transactions on the network. So, visit [here](#) and create an account. Save the mnemonic in a secure place (we would need it later). Instructions to add funds will be provided later in the tutorial.

Add .env File

Now we need an **Avalanche** wallet, where we would keep our funds, required for all the transactions on the network. Visit the [Avalanche Wallet](#) and create an account. While creating your account you will see your public address and mnemonic. This public address will be required to transfer funds. Save the **mnemonic** in a secure place, we will need it later. Instructions to add funds will be provided later in the tutorial.

Create a `.env` file in the project root folder. Please take a note that dot (.) is necessary for the `.env` file name. Now copy your Avalanche wallet's mnemonic in the `.env` file as shown below. In the `.env` file, **MNEMONIC** should be enclosed within double-quotes ("").

```
MNEMONIC=<avalanche-wallet-mnemonic>"
```

Never share or commit your `.env` file. It contains your credentials like `mnemonics`. Therefore, it is advised to add `.env` to your `.gitignore` file.

Now, the project setup is completed, and run the command given below in the project root folder to check whether we are on the same page or not.

```
npm start
```

Create Auction Smart Contract

Create an `Auction.sol` (sol stands for solidity) file inside the contracts directory and use the code given in this [file](#).

`Auction` is a solidity contract which enables us to view Auction details and correspondingly its minimum price. We will be accessing the deployed Auction contracts using their `address` and `ABI`. Each time when a new auction is created, the solidity code will be deployed to the blockchain.

Let's Understand this Contract in Detail

Users, Bids, Auctions and Analytics

```
// List of all auctions
Auction[] public auctions;

// Mapping for storing user info, bids and auction analytics
mapping (uint => User) public users;
mapping (uint => Bid[]) public bids;
mapping (uint => AuctionAnalytics) public auctionAnalytics;
```

The above block of code will declare public variables for storing user information, bids, auctions, and auction analytics. Have a look at the structures like `User`, `Bid`, etc of used in these variables.

Function to Check Registered User

```
// Public function to check the registration of users (public address)
function isRegistered(address _publicAddress) public view returns (uint256[2] memory) {
    uint256[2] memory result = [uint256(0), uint256(0)];
    for(uint i = 0; i < uId; i++) {
        if(_publicAddress == users[i].publicAddress) {
            result[0] = 1;
            result[1] = i;
            return result;
        }
    }
    return result;
}
```

This function takes the public address as its argument and returns an integer array with 2 elements - `isRegistered` at index **0** and `userId` at index **1**. If **0th** index is 1 then the user exists and vice-versa. And **** 1st index**** represents userId of the user. This function basically iterates over the mapping `users` to check if there is the required public address.

Auction Analytics

We have created a mapping for storing analytics like latest bid, highest bid, lowest bid, etc. for each auction. This mapping will map `auctionId` to `AuctionAnalytic` struct. Every time a new auction is created, we initialize its corresponding entry in the `AuctionAnalytics` map.

Updating Auction Analytics

```
// Private function to update auction analytics after the new bids
function updateAucionAnalytics(uint _aId, uint _latestBid) private {
```

```

        auctionAnalytics[_aId].latestBid = _latestBid;
        auctionAnalytics[_aId].auctionBidId = auctions[_aId].auctionBidId;
        if(_latestBid < auctionAnalytics[_aId].lowestBid) {
            auctionAnalytics[_aId].lowestBid = _latestBid;
        }
        if(_latestBid > auctionAnalytics[_aId].highestBid) {
            auctionAnalytics[_aId].highestBid = _latestBid;
        }
    }
}

```

Auction analytics needs to be updated every time there is a new bid. So, this function is called whenever a bid is created. It takes **auction id** and **latest bid amount** as its two arguments, and updates the analytics corresponding to the auction.

The rest of the functions are self-explanatory but are well commented for the readers to understand.

Create Migration Smart Contracts

Create the file `Migration.sol` inside of the contracts directory and paste the following code:

```

// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;

contract Migrations {
    address public owner = msg.sender;
    uint public last_completed_migration;

    modifier restricted() {
        require(
            msg.sender == owner,
            "This function is restricted to the contract's owner"
        );
    }

    function setCompleted(uint completed) public restricted {
        last_completed_migration = completed;
    }
}

```

`Migration.sol` smart contract manages the deployment of other contracts that we want to migrate to the chain.

Create a File for Migrating Smart Contracts

Create a new file in the `migrations` directory named `2_deploy_contracts.js`, and add the following block of code. This handles deploying the `Auction` smart contract to the blockchain.

```

const AuctionManager = artifacts.require("./AuctionManager.sol");

module.exports = function(deployer) {
    deployer.deploy(AuctionManager);
};

```

Compile Contracts with Truffle

If we have altered the code within our Solidity source files or made new ones (like `Auction.sol`), we need to run `truffle compile` in the terminal, from inside the project root directory.

The expected output would look similar:

```

Compiling your contracts...
=====
> Compiling ./contracts/Auction.sol
> Compiling ./contracts/Migrations.sol

> Artifacts written to /home/guest/blockchain/client/build/contracts
> Compiled successfully using:
  - solc: 0.8.0+commit.c7dfd78e.Emscripten clang

```

The compiled smart contracts are written as JSON files in the `/src/build/contracts` directory. These are the stored ABI and other necessary metadata - the artifacts.

ABI refers to Application Binary Interface, which is a standard for interacting with the smart contracts from outside the blockchain as well as contract-to-contract interaction. Please refer to the Solidity's documentation about ABIs [here](#) to learn more.

Fund the Account and Run Migrations on the C-Chain

During the deployment of the smart contract to the C-chain, deployment cost will be required. Already we have seen this inside `truffle-config.js` HDWallet Provider will help us in deploying on Fuji C-chain and deployment cost will be managed by the account whose mnemonic has been stored in the `.env` file. Therefore, we need to fund the account.

Fund Your Account

We need funds in our C-Chain address, as smart contracts are deployed on C-Chain, the Contract-Chain. This address can easily be found on the [Avalanche Wallet](#) dashboard. Avalanche network has 3 chains: X-Chain, P-Chain, and C-Chain. The address of all these chains can be found by switching tabs at the bottom of the division, where there is a QR code. So, switch to C-Chain, and copy the address. Now fund your account using the faucet link [here](#) and paste your C-Chain address in the input field. Refer to the below image, to identify the address section.



You'll need to send at least 135422040 nAVAX to the account to cover the cost of contract deployments. Here nAVAX refers nano-AVAX, that is a billionth of an AVAX, or simply 1 nAVAX = (1/1000,000,000) AVAX. Though funding through faucet would give you enough AVAX to run multiple deployments and transactions on the network.

Run Migration

All the required thing has been placed to run and now deploy the `Auction`.

```
truffle migrate --network fuji
```

The developer can deploy our contracts on a local network by executing Ganache (Truffle's local blockchain simulation) and using the command mentioned below

```
truffle migrate --network development
```

On successful execution of the above command, the developer may find the similar as mentioned below:

```
Starting migrations...
=====
> Network name: 'fuji'
> Network id: 1
> Block gas limit: 8000000 (0x7a1200)

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash: 0x094a9c0f12ff3158bcb40e266859cb4f34a274ea492707f673b93790af40e9e9
> Blocks: 0 Seconds: 0
> contract address: 0xb1f00d0Af6d5c864f86E6b96216e0a2Da111055
> block number: 40
> block timestamp: 1620393171
> account: 0x80599dd7F8c5426096FD189dcC6C40f47e8e3714
> balance: 39.71499696
> gas used: 173118 (0x2a43e)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00346236 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00346236 ETH

2_deploy_contracts.js
=====

Deploying 'AuctionManager'
-----
> transaction hash: 0xebc13fc6bbe250eea9151faf02bfe247ec497294acc84c9b8319ed609ced086
> Blocks: 0 Seconds: 0
> contract address: 0xf30D372A6911CCF6BBale84c3CEd51c0F3D7769
> block number: 42
> block timestamp: 1620393172
> account: 0x80599dd7F8c5426096FD189dcC6C40f47e8e3714
> balance: 39.69235442
> gas used: 1090212 (0x10a2a4)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.02180424 ETH
```

```
> Saving migration to chain.  
> Saving artifacts  
-----  
> Total cost: 0.02180424 ETH
```

```
Summary  
=====  
> Total deployments: 2  
> Final cost: 0.0252666 ETH
```

If you didn't create an account on the C-Chain you'll see this error:

```
Error: Expected parameter 'from' not passed to function.
```

If you didn't fund the account, you'll see this error:

```
Error: *** Deployment Unsuccessful***  
  
"Migrations" could not deploy due to insufficient funds  
* Account: 0x090172CD36e9f4906Af17B2C36D662E69f162282  
* Balance: 0 wei  
* Message: sender doesn't have enough funds to send tx. The upfront cost is: 1410000000000000000000000 and the sender's account  
only has: 0  
* Try:  
+ Using an adequately funded account
```

The information and ABI of the deployed contract are present in the `src/build/contracts` directory as `Auction.json`.

Building the User Interface

Our blockchain code, which will act as a backend for this application, is deployed on the chain and now we can code client-side for interacting with the contracts. We will be using Truffle Suite's **Drizzle** library for connecting our web app with blockchain. Drizzle makes the integration process very easy and scalable. It also provides a mechanism to **cache** a particular contract-call, so that, we can get a real-time update of the changes of data on the blockchain.

We will be using **React's context APIs** for facilitating our integration. Context APIs make the use of variables that are declared in the parent component very easy to access in the child components.

It is based upon the **Provider** and **Consumer** concepts. The **Provider** component contains the necessary logic and variables that need to be passed. Then this Provider component is wrapped around the components which want to access its variables. Every child component can access these variables. But in order to access it, we use **Consumer** API. This API will return the variables that are provided by the Provider component (only when called from its child). Look at the below codes for understanding it better.

In the `drizzleContext.js` file, **DrizzleProvider** is the provider component and **useDrizzleContext** is the consumer function. Look at the return statement of these functions. One is returning the Context Provider (provider) and the other the returning the values of the Context itself (consumer).

Drizzle Option Component

Create a file `drizzleOptions.js` inside the `drizzle-auction/client/src/` directory and paste the following code:

```
import AuctionManager from './build/contracts/AuctionManager.json';  
  
const drizzleOptions = {  
  contracts: [AuctionManager]  
}  
  
export default drizzleOptions;
```

The `drizzleOptions` constant contains the configuration like contracts we want to deploy, our custom web3 provider, smart contract events, etc. Here we just instantiating only the `AuctionManager` Smart contract.

Index Component

Inside the file `index.js` of `src` directory, paste the following code mentioned below:

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';  
  
import { Drizzle } from "@drizzle/store";  
import drizzleOptions from './drizzleOptions';  
import { DrizzleProvider } from './drizzleContext';  
  
const drizzle = new Drizzle(drizzleOptions);
```

```

ReactDOM.render(
  <DrizzleProvider drizzle={drizzle}>
    <App/>
  </DrizzleProvider>
,
  document.getElementById('root')
);

```

Importing `Drizzle` from `@drizzle/store` module will help in instantiating the drizzle according to our `drizzleOptions`. The following line is responsible for this action.

```
const drizzle = new Drizzle(drizzleOptions);
```

Then we wrap the `App` component inside the `DrizzleProvider`, so that, we can use extracted variables (see `drizzleContext.js`) inside App. We pass the `drizzle` object to the provider component because it will be required to extract other information from it.

Drizzle Context

Create a file `drizzleContext.js` inside the `drizzle-auction/client/src/` directory and paste the following code mentioned below:

```

import React, { createContext, useContext, useState } from "react";

const Context = createContext();

export function DrizzleProvider({ drizzle, children }) {
  const [drizzleVariables, setDrizzleVariables] = useState({
    initialized: false,
    state: null,
    web3: null,
    accounts: null,
    AuctionManager: null,
    subscriber: null
  });
  const unsubscribe = drizzle.store.subscribe(() => {
    const drizzleState = drizzle.store.getState();
    if (drizzleState.drizzleStatus.initialized) {
      const { web3, accounts } = drizzleState;
      const AuctionManager = drizzle.contracts.AuctionManager.methods;
      const subscriber = drizzleState.contracts.AuctionManager;
      setDrizzleVariables({
        state: drizzleState,
        web3,
        accounts,
        AuctionManager,
        subscriber,
        initialized: true
      });
    }
  });
  drizzleVariables.initialized && unsubscribe();

  return <Context.Provider value={{drizzle, drizzleVariables}}>{children}</Context.Provider>;
}

export function useDrizzleContext() {
  const context = useContext(Context);
  return context;
}

```

`DrizzleProvider` function takes `drizzle` as its argument and extracts other information like whether drizzle contracts are initialized or not, web3 info, account info, deployed contract's instance, etc. We need to `subscribe` to the drizzle's `store` for this information because these data are not fetched at once, and since we do not know when will we get these data, we subscribe to the store (where all data resides). Once the drizzle is initialized with contract data, we `unsubscribe` to store, so that, it will not re-render infinitely many times.

Drizzle State

```
const drizzleState = drizzle.store.getState();
```

This variable holds the state of the store which consists of data like web3 provider, account info, cached call information, etc. Cached calls are those contract calls for which we want real-time data from the blockchain. Whenever there is some change in our data on the blockchain, it gets notified in the `drizzle state` variable of the store.

AuctionManager

```
const AuctionManager = drizzle.contracts.AuctionManager.methods;
```

drizzle.contracts is an object which contains instances of all the deployed contracts which are added to drizzle (in drizzleOptions or manually). We are simply storing all the methods of this contract instance, so that, whenever we want to call a function or public identifier from this contract, we can simply use `AuctionManager.method_name().call()`

App Component

Now open `App.js` inside the `drizzle-auction/client/src/` directory and paste the following code as mentioned below:

```
import Auction from "./Auction";
import { useDrizzleContext } from "./drizzleContext";

function App() {
  const {drizzleVariables} = useDrizzleContext();

  if(!drizzleVariables.initialized) {
    return "Loading..."
  } else {
    return <Auction/>
  }
}

export default App;
```

`drizzleVariables.initialized` would ensure that, `Loading...` state is visible until Drizzle is ready for interaction.

Auction Component

Create a file `Auction.js` inside the `drizzle-auction/client/src/` directory, and paste the code given in this [file](#). This component deals with the entry-point of our application, where all the data like `userInfo`, `AuctionLists`, `AuctionDetails` etc. get generated.

In order to keep data fresh from the blockchain, Drizzle uses the caching mechanism. On our behalf Drizzle keeps track of every change on the blockchain. If there is any transaction involving our smart contracts, then it will notify our dapp.

We need to specifically define the calls which we want to monitor. Caching a particular method will provide cache keys (hash) to us. Each cached method is associated with a particular unique hash. Using this key, we can get live data from the blockchain, and the component will re-render anytime there is some new value associated with this call.

For example, in the above code, we used the following cache keys

```
const [cacheKeys, setCacheKey] = useState({
  uid: null,
  aid: null,
  showAuctions: null,
  isRegistered: null,
  auctionAnalytics: [null]
});
```

Suppose we want to cache `isRegistered` method, then this can be done using

```
const _isRegistered = AuctionManager?.isRegistered?.cacheCall(accounts[0]);
setCacheKey({
  isRegistered: _isRegistered
})
```

Once a method is cached, the Drizzle `store` would create a key-value pair representing hash-key and real-time data associated with this call. In the above program, this data is accessed using the `subscriber` variable as follows

```
const realTimeIsRegistered = subscriber?.isRegistered[cacheKeys?.isRegistered]?.value
```

In this component, we made a simple object of cached call variables named `cacheCall`, which implements the above code snippet. The cached version of `isRegistered` can be accessed as `cacheCalls.isRegistered`.

Auction List

Create a file `Auctionlist.js` inside the `drizzle-auction/client/src/` directory and use the code as given in this [file](#). This component deals with the management of the auction like creating a new bid, displaying the real-time auction analytics, etc. All these data are passed by its parent component i.e. `Auction.js` which manages the cache keys and calls.

Creating New Auctions

Create a file `CreateAuction.js` inside the `drizzle-auction/client/src/` directory and use the code as given in this [file](#). This component deals with creation of new Auctions, by submitting transactions on the network.

Starting the Application

Now go to the project root directory of the project, that is the `drizzle-auction` directory, and run the command `npm start`. The ReactJS server would start automatically. Visit <http://localhost:3000> to interact with the built dapp.

Don't forget to set up MetaMask with Avalanche Fuji testnet and also fund the account with Avalanche test tokens to upload files.

In the MetaMask extension, add a custom RPC by clicking at the network dropdown in the center of the extension. Fill in the details as shown in the below image.

Info	Value
Network Name	Avalanche Fuji
New RPC URL	https://api.avax-test.network/ext/bc/C/rpc
Chain ID	43113
Currency Symbol	AVAX-C
Block Explorer URL	https://testnet.snowtrace.io



Conclusion

We have successfully built a dapp through which we can organize auctions, bid in them and declare results, with both frontend and smart contracts. We have used the Drizzle library from Truffle Suite for integrating our frontend with the blockchain and to keep our data updated in real-time.

Next Steps

Our dapp currently has very minimalistic designs. We can use ConsenSys' Rimble UI library for adding modals for each transaction, add links to drip Avalanche's test tokens etc. which can help users to navigate through our dapp.

About the Author

This tutorial was created by [Raj Ranjan](#).

Auction.js

```
import React, { useState, useEffect } from "react";
import { useDrizzleContext } from './drizzleContext';
import FetchAuctions from './AuctionList';
import CreateAuction from './CreateAuction';

function Auction() {
    // Importing drizzle variables from drizzle context
    const { drizzleVariables } = useDrizzleContext();
    const { AuctionManager, subscriber, accounts } = drizzleVariables;

    // Setting up cache keys corresponding to cache calls
    const [cacheKeys, setCacheKey] = useState({
        uId: null,
        aId: null,
        showAuctions: null,
        isRegistered: null,
        auctionAnalytics: [null]
    });
    const [auctionAnalyticsCacheKey, setAuctionAnalyticsCacheKey] = useState(null);

    // Setting up cache calls for required functions
    const cacheCalls = {
        isRegistered: subscriber?.isRegistered[cacheKeys?.isRegistered]?.value,
        user: subscriber?.users[cacheKeys?.uId]?.value,
        aId: subscriber?.aId[cacheKeys?.aId]?.value,
        showAuctions: subscriber?.showAuctions[cacheKeys?.showAuctions]?.value,
        auctionAnalytics: []
    }

    for(let i = 0; i < cacheCalls.aId; i++) {
        cacheCalls.auctionAnalytics.push(subscriber?.auctionAnalytics[auctionAnalyticsCacheKey[i]]?.value)
    }

    const [isRegistered, setIsRegistered] = useState(false);
    const [userInfo, setUserInfo] = useState({
        id: null,
        name: null
    })
}
```

```

    });

// Initializing cache keys
useEffect(() => {
  const _auctionCacheKey = AuctionManager?.showAuctions?.cacheCall()
  const _aIdCacheKey = AuctionManager?.aId?.cacheCall()
  const _isRegistered = AuctionManager?.isRegistered?.cacheCall(accounts[0])
  setCacheKey({
    ...cacheKeys,
    showAuctions: _auctionCacheKey,
    aId: _aIdCacheKey,
    isRegistered: _isRegistered
  })
}, []);

useEffect(() => {
  let _auctionAnalyticsCacheKey = [];
  for(let i = 0; i < cacheCalls.aId; i++) {
    _auctionAnalyticsCacheKey.push(AuctionManager?.auctionAnalytics?.cacheCall(i))
  }
  setAuctionAnalyticsCacheKey(_auctionAnalyticsCacheKey)
}, [cacheCalls.aId]);

useEffect(() => {
  if(cacheCalls.isRegistered !== undefined && cacheCalls.isRegistered[0] == 1) {
    setIsRegistered(true);
    (async () => {
      const userInfo = await AuctionManager.users(cacheCalls.isRegistered[1]).call()
      setUserInfo({
        id: userInfo.userId,
        name: userInfo.name
      })
    })();
  } else {
    setIsRegistered(false)
  }
}, [cacheCalls.isRegistered]);

const createUser = async (name) => {
  await AuctionManager?.createUser(name)?.send({from: accounts[0]})
}

const [userName, setUserName] = useState("");

const handleUserNameChange = (event) => {
  setUserName(event.target.value)
}

const submitLogin = event => {
  event.preventDefault()
  createUser(userName)
}

const UserInfo = () => {
  return (
    <div>
      <label style={{color: "red"}}>ID: </label> {userInfo.id}
      <label style={{marginLeft: "50px", color: "green"}}>Name: </label> {userInfo.name} <br/><br/>
    </div>
  )
}

return (
  <div>
    <h1>Auctions</h1>
    {
      isRegistered
      ?
      <>
        <UserInfo/>
        <FetchAuctions cacheCalls={cacheCalls} userInfo={userInfo}/> <br/><br/>
        <CreateAuction/>
      </>
      :
      <form onSubmit={submitLogin}>
        <font color="red" font="2">This address is not yet registered!</font><br/><br/>
        <label>Address: </label><input disabled value={accounts[0]}/><br/><br/>
        <label>Name: </label><input key="1" value={userName} required onChange={handleUserNameChange}>
      </form>
    }
  </div>
)
}

```

```

placeholder="Enter your name"/>><br/><br/>
        <input type="submit" value="Register" />
    </form>
}
</div>
)
}

export default Auction;

```

AuctionList.js

```

import React, { useState } from "react";
import { useDrizzleContext } from './drizzleContext';

function FetchAuctions({cacheCalls, userInfo}) {
    const { drizzleVariables } = useDrizzleContext();
    const { AuctionManager, accounts } = drizzleVariables;

    const [bidPrices, setBidPrices] = useState(new Map([]))

    const createBid = async (id, bidPrice) => {
        await AuctionManager?.createBid(id, bidPrice).send({from: accounts[0]});
        clearBidPriceInput(id);
    }

    const submitNewBid = (event, id) => {
        event.preventDefault();
        createBid(id, bidPrices.get(id));
    }

    const handleBidPriceChange = (event, id) => {
        let _bidPrices = bidPrices;
        _bidPrices.set(id, event.target.value);
        setBidPrices(_bidPrices)
    }

    const clearBidPriceInput = (id) => {
        let _bidPrices = bidPrices;
        _bidPrices[id] = "";
        setBidPrices(_bidPrices)
    }

    const getAuctionAnalytics = () => {
        let auctionAnalytics = [];
        for(let i = 0; i < cacheCalls.aId; i++) {
            auctionAnalytics.push(cacheCalls.auctionAnalytics[i]);
        }
        return auctionAnalytics;
    }

    const allAuctions = cacheCalls.showAuctions;
    const auctionAnalytics = getAuctionAnalytics();
    return (
        <table border="1" style={{maxWidth: "800px", width: "90%}}>
            <tr>
                <td>Auction ID</td>
                <td>Auction Details</td>
                <td>Minimum Price</td>
                <td>Bid</td>
            </tr>
            {
                allAuctions !== undefined && allAuctions.map((auction, index) => (
                    <tr>
                        <td>
                            {auction.auctionId}
                        </td>
                        <td>
                            <b>{auction.name} <font size="2" color="green">{(auction.userId == userInfo.id && "(created by you")}</font></b><br/>
                            <font size="2">{auction.description}</font><br/>
                        </td>
                        <td>Total Bids</td>
                        <td>Latest Bid</td>
                        <td>Highest Bid</td>
                    </tr>
                ))
            }
        </table>
    )
}

export default AuctionList;

```

```

        <td>Lowest Bid</td>
    </tr>
    <tr>
        <td>(auctionAnalytics[index]?.auctionBidId)</td>
        <td>₹(auctionAnalytics[index]?.latestBid)</td>
        <td>₹(auctionAnalytics[index]?.highestBid)</td>
        <td>₹(auctionAnalytics[index]?.auctionBidId == 0 ? 0 : auctionAnalytics[index]?.lowestBid)</td>
    </tr>
</td>
<td>
    ₹{auction.msp}
</td>
<td>
    <form onSubmit={(event) => submitNewBid(event, auction.auctionId)} style={{margin: "10px"}}>
        <input required type="number" min={auction.msp} onChange={(event) => handleBidPriceChange(event, auction.auctionId)} placeholder="Enter your bid price"/><br/><br/>
        <input type="submit" value="Make Bid"/><br/><br/>
    </form>
</td>
</tr>
)
)
</table>
)
}

export default FetchAuctions;

```

CreateAuction.js

```

import React, { useState } from "react";
import { useDrizzleContext } from './drizzleContext';

function CreateAuction() {
    // Importing drizzle variables from drizzle context
    const { drizzleVariables } = useDrizzleContext();
    const { AuctionManager, accounts } = drizzleVariables;

    const createAuction = async ({title, description, msp}) => {
        await AuctionManager?.createAuction(title, description, msp)?.send({from: accounts[0]});
        setAuctionDetails({
            title: '',
            description: '',
            msp: ''
        })
    }

    const [auctionDetails, setAuctionDetails] = useState({
        title: '',
        description: '',
        msp: ''
    });

    const handleAuctionTitleChange = (event) => {
        setAuctionDetails({
            ...auctionDetails,
            title: event.target.value
        });
    }

    const handleAuctionDescriptionChange = (event) => {
        setAuctionDetails({
            ...auctionDetails,
            description: event.target.value
        });
    }

    const handleAuctionMspChange = (event) => {
        setAuctionDetails({
            ...auctionDetails,
            msp: event.target.value
        });
    }

    const submitNewAuction = (event) => {

```

```

        event.preventDefault();
        createAuction(auctionDetails);
    }

    return (
        <form onSubmit={submitNewAuction} style={{border: "1px black solid", maxWidth: "400px", padding: "10px"}}>
            <label>Title: </label><br/><input value={auctionDetails.title} onChange={handleAuctionTitleChange}/><br/><br/>
            <label>Description: </label><br/><textarea rows="4" value={auctionDetails.description} onChange={handleAuctionDescriptionChange}/><br/><br/>
            <label>MSP: </label><br/><input value={auctionDetails.msp} onChange={handleAuctionMspChange}/><br/><br/>
            <input type="submit" value="Create Auction" />
        </form>
    )
}

export default CreateAuction;

```

Create a Voting Dapp on Avalanche Using ReactJS

Introduction

We will generate [ReactJS](#) boilerplate code using `create-react-app`, which we will modify for our dapp frontend. React is a good choice for efficient, developer-friendly blockchain interactions. For the backend, [Solidity](#) smart contracts will be deployed to the Avalanche blockchain using [Truffle Suite](#).

Truffle Suite is a toolkit for launching decentralized applications (dapps) on Ethereum Virtual Machine (EVM) compatible blockchains like Avalanche. With Truffle you can write and compile smart contracts, build artifacts, run migrations and interact with deployed contracts. This tutorial illustrates how Truffle can be used with Avalanche's C-Chain, which is an instance of the EVM.

Prerequisites

- Basic familiarity with [NodeJS](#) and [npm](#).
- Basic familiarity with [ReactJS](#).
- Basic familiarity with [Avalanche](#) network, [Solidity](#) and [Truffle](#).

Requirements

- [NodeJS](#) >= 10.16 and [npm](#) >= 5.6 installed.
- [Truffle](#), which can be installed globally with `npm install -g truffle`
- [MetaMask](#) extension added to the browser.

Initializing the Working Directory

The client-side of our dapp is made using [ReactJS](#). Smart contracts will be made using [Solidity](#) language and will be deployed on the [Avalanche](#) network with [Truffle Suite](#). Therefore, we need to set up our working directory according to ReactJS and Truffle, to make the development process smoother.

Open a terminal and navigate to the directory where we will create the application. Usually, this will be inside our user home directory but can be located wherever is practical. On most Linux distributions this will change into `/home/`. On macOS it will be `/Users/`. On Windows the user directories are located in `C:\Users\`.

```
cd ~
```

Setting up the ReactJS Project

Create a new react app using `npx`. `npx` is an npm package runner (`x` probably stands for `eXecute`). The typical use is to download and run a package temporarily or for trials. Using `npx` to execute the package binaries for `create-react-app` will generate a new React app scaffold in the specified directory.

```
npx create-react-app avalanche-voting
```

Move to the newly created directory and install the basic dependencies.

```
cd avalanche-voting
npm install --save dotenv web3 @truffle/contract @truffle/hdwallet-provider
```

Open the file `index.html` file inside of the `public` directory and replace the existing code with the following HTML :

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Avalanche Elections</title>
    <link
        rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.0/dist/css/bootstrap.min.css"
    />
</head>

```

```
<body>
  <div id="root"></div>
</body>
</html>
```

Open the file `App.js` inside of the `src` directory and replace the existing code with the following code:

```
import React from "react";

// 1. Importing other modules

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      web3: null,
      account: null,
      mainInstance: null,
    };
  }

  componentDidMount() {
    this.init();
  }

  async init() {
    // 2. Load web3
    // 3. Load Account
    // 4. Load Smart-Contract instance
  }

  render() {
    return <div>Avalanche voting application</div>;
  }
}
export default App;
```

This `App` component has a constructor to declare and initialize the state properties. `web3` is an instance of the `MetaMask` provider for interacting with the Avalanche network, `account` is a user address and `mainInstance` is the instance of our smart contract.

Open the file `index.js` inside of the `src` directory and replace the existing code with the following code:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
);
```

React project setup is now complete.

Setting up the Truffle Project

Run the following command in the project root directory, to create a boilerplate for the Truffle project.

```
truffle init
```

This will set up the initial project structure. Solidity code will be stored in the `contracts` directory. Deployment functions written in JavaScript will be stored in the `migrations` folder. By default, the `/build/contracts` folder contains information about the compiled and deployed contract, like the ABI, in JSON format. These meta-files are commonly referred to as `artifacts`.

There is also 1 **config** file created by the above command, which is, **truffle-config.js**. In this file, there is a lot of information regarding how to deploy contracts, how to choose a network to deploy them, and many others. Therefore, we should preserve this file for reference. So, use the below command to make a copy of this file. This would create a copy named `truffle-config-default.js`.

```
cp truffle-config.js truffle-config-default.js
```

Now we will update the `truffle-config.js` file, with the information needed to deploy the smart contract on the Fuji test network. This file helps us in connecting to the Avalanche node, and we will require an Avalanche node's RPC url along with an Avalanche wallet mnemonic for deploying the contract on the network.

```

require("dotenv").config();
const HDWalletProvider = require("@truffle/hdwallet-provider");

// Account credentials from which our contract will be deployed
const MNEMONIC = process.env.MNEMONIC;

module.exports = {
  contracts_build_directory: "./src/build/contracts",
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*",
    },
    fuji: {
      provider: function () {
        return new HDWalletProvider({
          mnemonic: MNEMONIC,
          providerOrUrl: `https://api.avax-test.network/ext/bc/c/rpc`,
        });
      },
      network_id: "*",
      gas: 3000000,
      gasPrice: 470000000000,
      skipDryRun: true,
    },
  },
  compilers: {
    solc: {
      version: "0.8.0",
    },
  },
};

```

Note that we're setting the `gasPrice` and `gas` to the appropriate values for the Avalanche C-Chain. Here, you can see that we have used `contracts_build_directory` to change the default location of `artifacts` from the project root directory to the `src` folder. This is because React cannot access files that are present outside the `src` folder.

Get Avalanche Credentials

For deploying smart contracts we need two things: A node connected to the Avalanche network and an account with few AVAX. Avalanche connected node through RPC (Remote Procedure Call) is provided for free by the Avalanche Networks.

Now we need an Avalanche wallet, where we would keep our funds, required for all the transactions on the network. So, visit [here](#) and create an account. Save the mnemonic in a secure place (we would need it later). Instructions to add funds will be provided later in the tutorial.

Add .env File

Now we need a **Avalanche** wallet, where we would keep our funds, required for all the transactions on the network. Visit the [Avalanche Wallet](#) and create an account. While creating your account you will see your public address and mnemonic. This public address will be required to transfer funds. Save the **mnemonic** in a secure place, we will need it later. Instructions to add funds will be provided later in the tutorial.

Create a `.env` file in the project root folder. Please take a note that dot (.) is necessary for the `.env` file name. Now copy your Avalanche wallet's mnemonic in the `.env` file as shown below. In the `.env` file, **MNEMONIC** should be enclosed within double-quotes ("").

```
MNEMONIC=""
```

Never share or commit your `.env` file. It contains your credentials like mnemonics. Therefore, it is advised to add `.env` to your `.gitignore` file.

Our project setup is now complete.

To confirm that we are on the same page, run the following command in the project root folder.

```
npm start
```

It might take few seconds, to show output as in the image below.



In a browser, visit the URL of our running dapp: <http://localhost:3000>. If you followed the above steps, you would see the page as shown below.



Create Election Smart Contract

Create the file `Election.sol` (`.sol` stands for Solidity) inside of the `contracts` directory and use the code as given in this [file](#).

`Election` is a Solidity contract that lets us view the name and description, the candidates standing in an election, and vote for them. For this dapp, we will be accessing the deployed Election contracts using their `address` and `ABI`. This Solidity code is what will be deployed to the blockchain, each time we create a new election.

Let's understand this smart contract

The code for smart contract is everything within `contract Election { }`.

Basic fields about election - This block of code would be storing basic fields of each `Election` contract. Fields include `name` and `description`.

```
// Election details will be stored in these variables
string public name;
string public description;
```

Storing candidate details - The `Candidate` struct consists of the data fields `id`, `name` and `voteCount`. We will define a mapping between an unsigned integer (`uint`) and each instance of a Candidate. This will enable us to refer to each candidate by its index within the mapping - `candidates[n]`, where `n` is the corresponding `uint` value.

```
// Structure of candidate standing in the election
struct Candidate {
    uint256 id;
    string name;
    uint256 voteCount;
}

// Storing candidates in a map
mapping(uint256 => Candidate) public candidates;
```

Storing address of voters who have already voted and the number of candidates - `voters` is a mapping between the address of the voter and a boolean. In Solidity, the default boolean value is `false`, so if the return value of `voters(address)` is `false` we can understand that this address has not voted. `true` indicates that the account has voted already.

```
// Storing address of those voters who already voted
mapping(address => bool) public voters;

// Number of candidates in standing in the election
uint public candidatesCount = 0;
```

Constructor call and adding candidates to the election - When a smart contract is deployed on Avalanche, the first function to be called is the `constructor()` function. Whatever we want to initialize in our Solidity smart contract, we do it inside the `constructor()` function. We will be adding a name, description, and candidates to the election. Here, `addCandidate()` is a private function, so that it cannot be called publicly. This function takes `name` and `description` as a single array named `_nda` in the first argument and candidates' name as an array in the second argument.

```
// Setting of variables and data, during the creation of election contract
constructor (string[] memory _nda, string[] memory _candidates) public {
    require(_candidates.length > 0, "There should be atleast 1 candidate.");
    name = _nda[0];
    description = _nda[1];
    for(uint i = 0; i < _candidates.length; i++) {
        addCandidate(_candidates[i]);
    }
}

// Private function to add a candidate
function addCandidate (string memory _name) private {
    candidates[candidatesCount] = Candidate(candidatesCount, _name, 0);
    candidatesCount++;
}
```

Voting candidates in an election - We made a `vote()` function. It takes `candidateId` as an argument and increments the vote of the respective candidate. It requires two things, a voter should not have voted in the particular election by checking boolean across the `voters` mapping and `candidateId` should be a valid one, that is `0 <= candidateId < candidatesCount`.

```
// Public vote function for voting a candidate
function vote (uint _candidate) public {
    require(!voters[msg.sender], "Voter has already Voted!");
    require(_candidate < candidatesCount && _candidate >= 0, "Invalid candidate to Vote!");
    voters[msg.sender] = true;
    candidates[_candidate].voteCount++;
}
```

Create MainContract Smart Contract

Create the file `MainContract.sol` inside of the `contracts` directory and paste the following code:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.0;

import './Election.sol';

contract MainContract {
    uint public electionId = 0;
    mapping (uint => address) public Elections;

    function createElection (string[] memory _nda, string[] memory _candidates) public {
        Election election = new Election(_nda, _candidates);
        Elections[electionId] = address(election);
        electionId++;
    }
}
```

`MainContract.sol` is the entry point of our dapp. To create a new election, we need to call the `createElection()` function from this deployed contract. It will maintain the total number of election contracts deployed, their address on the network and will also help in deploying them. We also [import](#) `Elections.sol`.

Here `electionId` is used for assigning ID's to each election that a user creates and is incremented for using it while creating the next election. Also, `Elections` is a public mapping between `electionId` and the address of the deployed election contract.

```
uint public electionId = 0;
mapping (uint => address) public Elections;
```

We have made a `createElection()` function which will be used to deploy our `Election` smart contract. This function takes `name` and `description` as a single array named `_nda` in the first argument and candidates' name as an array in the second argument.

```
function createElection (string[] memory _nda, string[] memory _candidates) public {
    Election election = new Election(_nda, _candidates);
    Elections[electionId] = address(election);
    electionId++;
}
```

The `Election` contract is deployed on the network using the `new` keyword, which deploys the contract, initializes the contract's variables, runs the `constructor()` function and returns the `address` of the newly deployed contract to the caller. Then the address is stored in the `Elections` mapping. Once the election contract is deployed successfully, `electionId` is incremented.

Create a File for Migrating Smart Contracts

Create a new file in the `migrations` directory named `2_deploy_contracts.js`, and add the following block of code. This handles deploying the `MainContract` and `Election` smart contract to the blockchain.

```
const MainContract = artifacts.require("MainContract");

module.exports = function (deployer) {
    deployer.deploy(MainContract);
};
```

We are deploying only the `MainContract`, because the `Election` contract will be deployed by the `MainContract` itself during the runtime, using the function `createElection()`.

Compile Contracts with Truffle

If we have altered the code within our Solidity source files or made new ones (like `Elections.sol`), we need to run `truffle compile` in the terminal, from inside the project root directory.

The expected output would look similar:

```
Compiling your contracts...
=====
> Compiling ./contracts/Election.sol
> Compiling ./contracts/MainContract.sol
> Compiling ./contracts/Migrations.sol

> Artifacts written to /home/guest/blockchain/avalanche-voting/build/contracts
> Compiled successfully using:
  - solc: 0.8.0+commit.c7dfd78e.Emscripten clang
```

The compiled smart contracts are written as JSON files in the `/src/build/contracts` directory. These are the stored ABI and other necessary metadata - the artifacts.

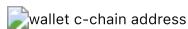
`ABI` refers to Application Binary Interface, which is a standard for interacting with the smart contracts from outside the blockchain as well as contract-to-contract interaction. Please refer to the Solidity's documentation about ABIs [here](#) to learn more.

Fund the Account and Run Migrations on the C-Chain

When deploying smart contracts to the C-Chain, it will require some deployment cost. As you can see inside `truffle-config.js`, HDWallet Provider will help us in deploying on Fuji C-chain and deployment cost will be managed by the account whose mnemonic has been stored in the `.env` file. Therefore we need to fund the account.

Fund Your Account

We need funds in our C-Chain address, as smart contracts are deployed on C-Chain, the Contract-Chain. This address can easily be found on the [Avalanche Wallet](#) dashboard. Avalanche network has 3 chains: X-Chain, P-Chain and C-Chain. The address of all these chains can be found by switching tabs at the bottom of the division, where there is a QR code. So, switch to C-Chain, and copy the address. Now fund your account using the faucet link [here](#) and paste your C-Chain address in the input field. Refer to the below image, to identify the address section.



You'll need to send at least 135422040 nAVAX to the account to cover the cost of contract deployments. Here nAVAX refers nano-AVAX, which is a billionth of an AVAX or simply 1 nAVAX = (1/1000,000,000) AVAX. Though funding through faucet would give you enough AVAX to run multiple deployments and transactions on the network.

Run Migrations

Now everything is in place to run migrations and deploy the `MainContract`:

```
truffle migrate --network fuji
```

This might take a while depending upon your internet connection or traffic on the network.

Note - For development purpose, we may deploy our contracts on the local network, by running Ganache (Truffle's local blockchain simulation) and using the command

```
truffle migrate --network development
```

On successful execution of this command, you should see:

```
Starting migrations...
=====
> Network name: 'fuji'
> Network id: 1
> Block gas limit: 800000 (0x7a1200)

1_initial_migration.js
=====

Deploying 'Migrations'
-----
> transaction hash: 0x094a9c0f12ff3158bcb40e266859cb4f34a274ea492707f673b93790af40e9e9
> Blocks: 0 Seconds: 0
> contract address: 0x0b1f00d0Af6d5c864f86E6b96216e0a2Da111055
> block number: 40
> block timestamp: 1620393171
> account: 0x80599dd7F8c5426096FD189dccC6C40f47e8e3714
> balance: 39.71499696
> gas used: 173118 (0x2a43e)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.00346236 ETH

> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.00346236 ETH

2_deploy_contracts.js
=====

Deploying 'MainContract'
-----
> transaction hash: 0xebef13fc6bbe250eea9151faf02bfe247ec497294acc84c9b8319ed609ced086
> Blocks: 0 Seconds: 0
> contract address: 0xf30D372A6911CCF6BBale84c3CEd51cC0F3D7769
> block number: 42
> block timestamp: 1620393172
> account: 0x80599dd7F8c5426096FD189dccC6C40f47e8e3714
```

```
> balance: 39.69235442
> gas used: 1090212 (0x10a2a4)
> gas price: 20 gwei
> value sent: 0 ETH
> total cost: 0.02180424 ETH
```

```
> Saving migration to chain.
> Saving artifacts
-----
> Total cost: 0.02180424 ETH
```

```
Summary
=====
> Total deployments: 2
> Final cost: 0.0252666 ETH
```

If you didn't create an account on the C-Chain you'll see this error:

```
Error: Expected parameter 'from' not passed to function.
```

If you didn't fund the account, you'll see this error:

```
Error: *** Deployment Failed ***

"Migrations" could not deploy due to insufficient funds
  * Account: 0x090172CD36e9f4906Af17B2C36D662E69f162282
  * Balance: 0 wei
  * Message: sender doesn't have enough funds to send tx. The upfront cost is: 14100000000000000000 and the sender's account
only has: 0
  * Try:
    + Using an adequately funded account
```

The information and ABI of the deployed contract are present in the `src/build/contract` directory as `Election.json`. Information like contract address, network info etc. could be found here.

Building the User Interface

We have already set up our React project directory. The client-side files to interact with the Avalanche blockchain are present in the `src` directory. First, we will make a ReactJS component with a couple of functions to connect our browser with the Avalanche network. These functions will be kept in a separate file named `BlockchainUtil.js`.

BlockchainUtils Component

Create the file `BlockchainUtil.js` inside of the project `src` directory and paste the following code:

```
import React from "react";
import Web3 from "web3";
import TruffleContract from "@truffle/contract";

export class GetWeb3 extends React.Component {
  async getWeb3() {
    let web3 = window.web3;

    if (typeof web3 !== "undefined") {
      // Setup Web3 Provider
      this.web3Provider = web3.currentProvider;
      this.web3 = new Web3(web3.currentProvider);
      return this.web3;
    } else {
      this.isWeb3 = false;
    }
  }
}

export class GetContract extends React.Component {
  async getContract(web3, contractJson) {
    // Setup Contract
    this.contract = await TruffleContract(contractJson);
    this.contract.setProvider(web3.currentProvider);
    return await this.contract.deployed();
  }
}

export class GetAccount extends React.Component {
```

```

    async getAccount(web3) {
      return await web3.eth.getAccounts();
    }
}

```

Updating App.js

`App.js` is the entry point of our React application. Therefore, we will need to update `App.js` with the components which we want to show in our application. As we move ahead and build components, we will update `App.js` and import the components so that we can make use of their functionality.

So, now add the following line under the `//Importing...` section of `App.js` to import `BlockchainUtil.js` module.

```

// 1. Importing other modules
import { GetWeb3, GetContract, GetAccount } from "./BlockchainUtil";

```

Paste the following code inside the `init()` function of `App.js`

```

// 2. Load web3
const Web3 = new GetWeb3();
this.web3 = await Web3.getWeb3();
this.setState({ web3: this.web3 });

// 3. Load Account
const Account = new GetAccount();
this.account = await Account.getAccount(this.web3);
this.setState({ account: this.account[0] });

// 4. Load Contract
const Contract = new GetContract();
this.mainInstance = await Contract.getContract(this.web3, contractJson);
this.setState({ mainInstance: this.mainInstance });

```

CreateElection Component

Now let's make a component that will create new elections using our deployed smart contract. Create the file `CreateElection.js` inside of the project `src` directory and use the code as given in this [file](#). The code is commented to draw attention to the important parts.

ActiveElections Component

Create the file `ActiveElections.js` inside of the project `src` directory and use the code as given in this [file](#).

VoteModal Component

In the above component `ActiveElections.js`, we have used a component called `VoteModal` which contains the candidate details and a button to cast a vote. Now we will make this component available by creating a file named `VoteModal.js` inside the `src` directory. Use the code as present in this [file](#).

Integrating Components Into App.js

Now we need to update our `App.js` file with all the components that we have made so far.

Import Modules - First import all the modules and components into the `App.js` file by appending the following code under the `// 1. Importing other modules` section.

```

// 1. Importing other modules
import {
  BrowserRouter as Router,
  Route,
  Link,
  Redirect,
} from "react-router-dom";
import CreateElection from "./CreateElection";
import ActiveElections from "./ActiveElections";
import contractJson from "./build/contracts/MainContract.json";

```

Load components - Inside the `<div> return()` function in `App.js`, replace the sample text (`Avalanche evoting`) with the code of the following components.

```

// For routing through the react application
<Router>
  {/* Default route to ActiveElections component */}
  <Route path="/" exact>
    <Redirect to="/active" />
  </Route>

  {/* Navbar */}
  <nav
    className="navbar navbar-dark shadow"

```

```

    style={{
      backgroundColor: "#1b2021",
      height: "60px",
      color: "white",
      marginBottom: "50px",
    }}
  >
  {/* Link to Active election page (nav-header) */}
  <Link to="/active">
    <b style={{ cursor: "pointer", color: "white" }}>Avalanche Elections</b>
  </Link>

  {/* Account address on the right side of the navbar */}
  <span style={{ float: "right" }}>{this.state.account}</span>
</nav>

 {/* Route to CreateElection page */}
{
  <Route
    path="/createElection"
    exact
    component={() => <CreateElection account={this.state.account} />}
  />
}

 {/* Route to Active election page */}
{
  <Route
    path="/active"
    exact
    component={() => <ActiveElections account={this.state.account} />}
  />
}
</Router>

```

We have used few other dependencies which we didn't install earlier. So, run the following command in the terminal of your project directory.

```
npm install --save rimble-ui react-router-dom --force
```

Rimble UI library comes with a peer dependency of `react@16.9.0` which is not the latest version of React that we are using which is `react@17.0.2`. Running the `npm install` command without the `--force` tag would cause an unable to resolve dependency tree conflict. Thus, the `--force` tag is used to override any type of conflicts and proceeds with the installation anyway. Another way to resolve this conflict is by using the `--legacy-peer-deps` tag instead, but this would ignore all peer dependencies which we do not require, as the conflict is only between `react` and `rimble-ui`.

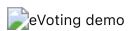
Now go to the project root directory, the `avalanche-voting` directory, and run the command `npm start`. The ReactJS server would start automatically. Visit <http://localhost:3000> in a browser to interact with the dapp frontend.

Don't forget to set up MetaMask with Fuji testnet and also fund the account with Fuji C-Chain test tokens to vote. In the MetaMask extension, add a custom RPC by clicking at the network dropdown in the centre of the extension. Fill in the details as shown below

Info	Value
Network Name	Avalanche Fuji
New RPC URL	https://api.avax-test.network/ext/bc/C/rpc
Chain ID	43113
Currency Symbol	AVAX-C
Block Explorer URL	https://testnet.snowtrace.io

Conclusion

You have successfully built a full-fledged e-voting dapp with advanced features like creating custom elections, voting in them and deployed the smart contract on the Fuji test network using Truffle Suite. Along with that, we have also built the client-side application using ReactJS for interacting with the network. From this tutorial, you have learned not only how to make and deploy smart contracts but also how to integrate ReactJS with the blockchain using Truffle Suite.



What's Next?

Now that we have built and deployed a voting dapp, we can make new elections with a title and description, and vote on them separately. Some recommended features to add would be the ability to add start and end dates for an election, declaring the winner after the election has ended, or UI enhancements like modal windows and styled buttons.

About the Author

This tutorial is created by [Raj Ranjan](#).

ActiveElections.js

```
import React, { Component } from "react";
import { Loader } from "rimble-ui";
import { Link } from "react-router-dom";

import App from "./App";
import ElectionJSON from "./build/contracts/Election.json";
import VoteModal from "./VoteModal";

// Election component for organising election details
let Election = (props) => (
  <tr>
    <td>(props.election.electionId)</td>

    <td>
      {props.election.electionName} <br />
      <font className="text-muted" size="2">
        <b>{props.election.electionDescription}</b>
      </font>
      <br />
      <font className="text-muted" size="2">
        {props.election.electionAddress}
      </font>
    </td>
  </td>
  <td style={{ textAlign: "center" }}>(props.candidateComponent)</td>

  <td style={{ textAlign: "center" }}>
    {!props.election.hasVoted ? (
      // Vote Modal would be mounted if the user has not voted
      <VoteModal election={props.election} candidates={props.candidates} />
    ) : (
      <font size="2" color="green">
        You have voted!
      </font>
    )}
  </td>
</tr>
);

// Candidate component for organising candidate details of each candidate
let Candidates = (props) => (
  <font size="2">
    <b>{props.name}</b> ({props.voteCount}) <br />
  </font>
);

// ActiveElections component would fetch and display all the elections deployed by the MainContract.sol
class ActiveElections extends Component {
  constructor(props) {
    super(props);
    this.state = {
      data: [],
      loading: false,
    };
  }

  // To store App.js instance
  app = null;

  // Connect application with Metamask and create smart-contract's instance
  async init() {
    this.app = new App();
    await this.app.init();
    await this.loadData();
  }

  loader = false;

  componentDidMount() {
    this.init();
  }
}
```

```

}

loadData = async () => {
  this.setState({ loading: true });

  // electionId maps to total elections created
  let eCount = await this.app.mainInstance.electionId();
  let elections = [], electionDetails = [], electionComponents = [];

  // Election details of every election created by MainContract
  for (let i = 0; i < eCount; i++) {
    elections[i] = await this.app.mainInstance.Elections(i);
    let election = await new this.app.web3.eth.Contract(ElectionJSON.abi, elections[i]);

    electionDetails[i] = [];

    // Account address of the voter
    electionDetails[i].account = this.app.account[0];

    // Each contract's instance
    electionDetails[i].contractInstance = election;

    // Address of each election contract
    electionDetails[i].electionAddress = elections[i];

    // Boolean indicating whether the contract address has voted or not
    electionDetails[i].hasVoted = await election.methods.voters(this.app.account[0]).call();

    // Name of the election
    electionDetails[i].electionName = await election.methods.name().call();

    // Description of the election
    electionDetails[i].electionDescription = await election.methods.description().call();

    // Election id
    electionDetails[i].electionId = i;

    // Organising candidates into components
    let candidatesCount = await election.methods.candidatesCount().call();
    let candidates = [], candidateComponents = [];
    candidates[i] = [];
    candidateComponents[i] = [];

    for (let j = 0; j < candidatesCount; j++) {
      candidates[i].push(await election.methods.candidates(j).call());
      candidateComponents[i].push(
        <Candidates
          name={candidates[i][j][1]}
          voteCount={candidates[i][j][2]}
        />
      );
    }
  }

  // Saving the electionDetails in the form of a component
  electionComponents[i] = (
    <Election
      election={electionDetails[i]}
      candidates={candidates[i]}
      candidateComponent={candidateComponents[i]}
    />
  );
}

this.setState({
  data: electionComponents,
  loading: false,
});
};

render() {
  return (
    // Simple container to store table with election data
    <div className="container">
      <div style={{ float: "right", marginBottom: "10px" }}>
        
      </div>
    </div>
  );
}

```

```

        />
      <Link to="/createElection">
        
      </Link>
    </div>

    <table className="table table-hover table-bordered">
      <thead>
        <tr>
          <th style={{ width: "120px" }}>Election ID</th>
          <th>Election Name</th>
          <th style={{ textAlign: "center" }}>Candidates</th>
          <th style={{ textAlign: "center" }}>Vote</th>
        </tr>
      </thead>

      <tbody>{this.state.data}</tbody>
    </table>

    <center>{this.state.loading ? <Loader size="40px" /> : <></>}</center>
  </div>
);
}
}

export default ActiveElections;

```

CreateElection.js

```

import React, { Component } from "react";
import App from "./App";

class CreateElection extends Component {
  constructor(props) {
    super(props);

    this.onChangeElectionName = this.onChangeElectionName.bind(this);
    this.onChangeDescription = this.onChangeDescription.bind(this);
    this.onSubmit = this.onSubmit.bind(this);

    // These state variables would maintain inputs of the form
    this.state = {
      electionname: "",
      description: "",
      candidates: [],
    };
  }

  // To store App.js instance
  app = null;

  // Connect application with Metamask and create smart-contract's instance
  async init() {
    this.app = new App();
    await this.app.init();
  }

  componentDidMount() {
    this.init();
  }

  onChangeElectionName(e) {
    this.setState({
      electionname: e.target.value,
    });
  }

  onChangeDescription(e) {
    this.setState({
      description: e.target.value,
    });
  }
}

```

```

// Function to be called when the form is submitted
async onSubmit(e) {
  e.preventDefault();

  // Structuring Election details from the form before submitting transaction to the smart-contract
  const electionDetails = {
    electionname: this.state.electionname,
    description: this.state.description,
    candidateObjects: document.getElementsByName("candidate").values(),
    candidates: []
  };

  let i = 0;

  for (let value of electionDetails.candidateObjects) {
    electionDetails.candidates[i] = value.value;
    i++;
  }

  // Making transaction to the MainContract instance, for creating a new election
  await this.app.mainInstance.createElection(
    [electionDetails.electionname, electionDetails.description],
    electionDetails.candidates,
    { from: this.app.account[0] }
  );

  window.location = "/active";
}

render() {
  return (
    <div className="container card">
      <h3>Create New Election</h3>

      /* New Election Form */
      <form onSubmit={this.onSubmit}>
        <div className="form-group">
          <label>Name</label>
          <input
            type="text"
            required
            className="form-control"
            placeholder="Enter election name"
            onChange={this.onChangeElectionName}
          />
        </div>

        <div className="form-group">
          <label>Description</label>
          <textarea
            type="text"
            required
            className="form-control"
            placeholder="Describe your Election here"
            onChange={this.onChangeDescription}
          ></textarea>
        </div>

        <table>
          <tr>
            <td id="1" className="form-group">
              <label>Candidate 1</label>
            <td>
              <input
                type="text"
                required
                className="form-control"
                placeholder="Candidate Name"
                name="candidate"
              />
            </td>

            <br />
            <label>Candidate 2</label>
            <td>
              <input
                type="text"
              
```

```

        required
        className="form-control"
        placeholder="Candidate Name"
        name="candidate"
      />
    </td>
  </td>
</tr>
</table>

<br />

<div>
  <button
    className="btn btn-success grid-item"
    style={{ width: 100 }}
    type="submit"
  >
    Submit
  </button>
</div>

<br />
</form>
</div>
);
}
}

export default CreateElection;

```

VoteModal.js

```

import React, { useState } from "react";
import { Box, Flex, Modal, Button, Text, Card, Radio, Field, Loader } from "rimble-ui";

// Data like election and candidate details will be passed in the props by ActiveElections.js (parent)
function VoteModal(props) {
  // These are React Hooks and are used only for UX like opening and closing of Voting Modal and loaders
  const [isOpen, setIsOpen] = useState(false);
  const [loading, isLoading] = useState(false);

  // This Hook will be used to maintain the selected candidate ID by a voter
  const [cid, changeCid] = useState(0);

  const closeModal = (e) => {
    e.preventDefault();
    setIsOpen(false);
  };

  const openModal = (e) => {
    e.preventDefault();
    setIsOpen(true);
  };

  const onRadioChange = (e) => {
    changeCid(e.target.value);
  };

  // vote() function would be used to transact a vote
  const vote = async (eid) => {
    isLoading(true);
    await props.election.contractInstance.methods.vote(cid).send({ from: props.election.account });
    isLoading(false);
  };

  let candid = [],
    candidVote = [];
  for (let i = 0; i < props.candidates.length; i++) {
    let candidDetail = props.candidates[i][1] + " (" + props.candidates[i][2] + ")";
    candid.push(
      <Radio
        name="candidate"
        key={i}
    
```

```

        label={candidDetail}
        my={2}
        value={props.candidates[i][0]}
        onChange={onRadioChange}
      />
    );
  }

  return (
    // This is a rimble-ui builtin modal for triggering vote() function
    <Box className="App" p={0}>
      <Box>
        <Button onClick={openModal}>Vote</Button>

        <Modal isOpen={isOpen}>
          <Card width={"420px"} p={0}>
            {/* Close icon to close the modal */}
            <Button.Text
              icononly
              icon="Close"
              color={"moon-gray"}
              position={"absolute"}
              top={0}
              top={0}
              right={0}
              mt={3}
              mr={3}
              onClick={closeModal}
            />

            {/* List of candidates with their vote count */}
            <Box p={4} mb={3}>
              <h3>{props.election.electionName}</h3>
              <Field label="Choose candidate from below">{candid}</Field>
            </Box>

            {/* Vote button to cast a vote */}
            <Flex
              px={4}
              py={3}
              borderTop={1}
              borderColor="#E8E8E8"
              justifyContent="flex-end"
            >
              {loading ? (
                <Loader size="40px" />
              ) : (
                <Button.Outline
                  onClick={() => {
                    vote(props.election.electionId);
                  }}
                >
                  Vote
                </Button.Outline>
              )}
            </Flex>
          </Card>
        </Modal>
      </Box>
    );
}

export default VoteModal;

```

Introduction

In this tutorial we will cover the use of Hardhat, a powerful EVM development tool, together with Avalanche. More specifically, we will cover how to configure it to work with Avalanche C-Chain and how to use the Hardhat `fork` mechanism to test your DeFi dApps.

Hardhat ?

Hardhat is an **Ethereum development environment for professionals**. It was developed for Ethereum, but since lots of other blockchains reuse the EVM you can apply Hardhat on those as well !

In short, it helps you in all the important steps of smart contract development. From compiling, deploying, and testing your Solidity code. It has even a functionality to let you use 'console.log' in your smart contract's code!

It's not the purpose of this tutorial to go over all those functionalities (maybe in another tutorial, why not :)), so here is a few links if you want to know more about it:

- [Getting started](#)
- [Hardhat](#)

What Is the 'Fork' Functionality ?

So let's get back to the core of this tutorial : Hardhat fork mechanism.

In order to make you realize the importance of this functionality, let me give you an example:

Let's say you have a simple contract `Swapper.sol`. It has a function `swap` that once called with the appropriate parameters will swap for you some Wavax tokens into another ERC20 tokens listed on a DEX. For the sake of this tutorial we will use [Pangolin](#)

The flow of it would be:

1. You send a call to Swapper's swap function
2. Swapper use Pangolin's router `swapExactTokensForTokens` function, it will find the appropriate Pair contract address to call
3. Pangolin's router call a Pair contract to make the swap

Notice how it requires calls to external contracts.

If you want to test your Swapper `swap` function you then need to set up in your test environment :

- 2 ERC20 contracts
- Pangolin Factory
- Pangolin router's contract
- A Pair contract (PGL) using your 2 previously created ERC20.
- And all this with the appropriate constructor parameters, linking all this together. Doable but it'll require some time to set up all this properly.

So, what if I told you that we could avoid all of this and jump directly to the step where you create tests for your smart contracts.

That's where 'Hardhat fork' is coming in handy.

With this you can simply make a copy of the state of an EVM chain (in our case the C-Chain) and use it directly in your tests ! With all contract, addresses balance available for you to use.

So in our case we would not have to deploy all the relevant Pangolin's contract, we could just use the one deployed on the Mainnet and test your smart contract without much hassle.

Step by Step Explanation

Smart Contract Overview

So first let's get over quickly the solidity code that we will use:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.4;

interface IRouter {
    function swapExactTokensForTokens(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline)
    external returns (uint[] memory amounts);
}

interface IERC20 {
    function transferFrom(address from, address to, uint value) external;
    function approve(address to, uint value) external returns (bool);
}

/**
 * I would not recommend you to use this code as it is. It is really simple and stripped of some basic security checks.
 */
contract Swapper {

    address private wavax; // Address of the ERC20 Wrapped Avax
    address private router; // Address of the 'Uniswap-like' router contract

    constructor(address _wavax_address, address _router){
        wavax = _wavax_address;
        router = _router;
    }

    /**
     * This function will perform a swap on Pangolin. The pair must be a WAVAX-ERC20 pair.
     * @param amountOutMin Minimum amount of token that we want to get after our swap.
     * @param path Array of tokens' address
     */
}
```

```

    * @param pair Address of the liquidity pair we will use in this swap
    * @param deadline Not relevant for avalanche, just pass timestamp that is in the future
    */
    function swap(uint256 amountOutMin, address[] calldata path, address pair, uint256 amountIn, uint256 deadline) external {
        // We transfer the wavax from the user (msg.sender) to this contract.
        IERC20(wavax).transferFrom(msg.sender, address(this), amountIn);
        // We approve the router as a spender for our Wavax.
        IERC20(wavax).approve(router, amountIn);
        // We do the swap using the router.
        IRouter(router).swapExactTokensForTokens(amountIn, amountOutMin, path, msg.sender, deadline);
    }
}

```

Hardhat Configuration

As you can see we use some external contract (Pangolin router). Meaning that if you want to test this code... you'll have to mock/recreate this router and all the contracts that this router use ... Kinda annoying right ?

Thanks to Hardhat, we can make our life easier and skip it altogether.

First we need to configure Hardhat. If you look in `hardhat.config.ts` you'll see this :

```

const config: HardhatUserConfig = {
  defaultNetwork: "hardhat",
  solidity: "0.8.4",
  networks: {
    hardhat: {
      chainId: 43114,
      gasPrice: 225000000000,
      forking: {
        url: "https://api.avax.network/ext/bc/C/rpc",
        enabled: true,
      },
    },
    fuji: {
      chainId: 43113,
      gasPrice: 225000000000,
      url: "https://api.avax-test.network/ext/bc/C/rpc",
      accounts: [
        PK_TEST
      ]
    }
  },
  typechain: {
    outDir: "typechain",
    target: "ethers-v5",
  },
};

```

The most interesting portion of code here is the `network` part. That's where you configure networks that you want to use with your project. As you can see above we have defined two networks for this tutorial:

- `hardhat`, which is also the `defaultNetwork`.
- `fuji`, which is pointing to Fuji testnet.

Note that you can put multiple network definition, one of it is considered as the 'default' one. Meaning that when you are using `npx hardhat test`, it'll use the default network. If you want to run the test on another network than the default, you can use this variation of the command : `npx hardhat test --network fuji`

Now let's focus on the `hardhat` one .

```

hardhat: {
  chainId: 43114,
  gasPrice
:
  225000000000,
  forking
:
  {
    url: "https://api.avax.network/ext/bc/C/rpc",
    enabled
:
    true,
    blockNumber
:
    2975762
  }
,

```

- ```

}

'

 - chainId is set with to the Mainnet value, as seen here.
 - gasPrice is a dynamic value on Avalanche's C-Chain (see this post for more information). For test purposes we can use a fixed value (225 nAvax)
 - forking is where you configure the parameter of the fork.
 - url here we see that we point to the Ava labs API endpoint This could be your local node, as long as it is running as full archive node. Hardhat will take care of getting the state of the C-Chain from this node and start a local development network on which you'll be able to deploy and test your code.
 - blockNumber Specify at which block Hardhat will create a fork. It is optional so if you don't set it, the default behaviour would be to fork the C-Chain at the latest known block. Now since you want to be able to run your tests in a deterministic manner, I recommend you to specify a specific block number.
```

If you want to see all configurations options, please go check the [official documentation](#) for this feature.

## Tests Overview

So we went over the Solidity code, the Hardhat configuration. Now let's have a look at how to create a test using Hardhat.

Now let's have a look at the test code.

Testing with Hardhat is fairly simple, lots of things are abstracted away.

Let's first have a first look at the test I've written for our Swapper contract, no worries we will dissect it a bit later.

```

import { ethers } from "hardhat";
import * as dotenv from "dotenv";
import { SignerWithAddress } from "hardhat-deploy-ethers/src/signers";
import { BigNumber } from "ethers";
import { Swapper, IWAVAX } from "../typechain";

dotenv.config();

const AVALANCHE_NODE_URL: string = process.env.AVALANCHE_MAINNET_URL as string;
const WAVAX_ADDRESS: string = process.env.WAVAX_ADDRESS as string;
const PNG_ADDRESS = "0x60781C2586D68229fde47564546784ab3fACA982"

describe("Swappity swap", function () {
 let swapper: Swapper;
 let account1: SignerWithAddress;

 beforeEach(async function () {
 await ethers.provider.send(
 "hardhat_reset",
 []
);
 [
 {
 forking: {
 jsonRpcUrl: AVALANCHE_NODE_URL,
 blockNumber: 2975762,
 },
 },
],
);
 let accounts = await ethers.getSigners()

 // @ts-ignore
 account1 = accounts[0]

 // Here we get the factory for our Swapper contract and we deploy it on the forked network
 const swapperFactory = await ethers.getContractFactory("Swapper")
 swapper = await swapperFactory.deploy(process.env.WAVAX_ADDRESS as string, "0xE54Ca86531e17Ef3616d22Ca28b0D458b6C89106");
});

it("should swap wavax for png", async function () {
 // We get an instance of the wavax contract
 const wavaxTokenContract = await ethers.getContractAt("IWAVAX", WAVAX_ADDRESS)
 // @ts-ignore
 const pngTokenContract = await ethers.getContractAt("IWAVAX", PNG_ADDRESS)
 //makes sure owner has enough WAVAX balance
 if ((await wavaxTokenContract.balanceOf(account1.address)).lt("1000000000000000000000000000000")) {
 await wavaxTokenContract.deposit({
 value: BigNumber.from("1000000000000000000000000000000")
 .sub(await wavaxTokenContract.balanceOf(account1.address))
 })
 }
})

```

```

// We tell Wavax contract that we are cool with Swapper contract using our Wavax on our behalve
await wavaxTokenContract.approve(swapper.address, ethers.constants.MaxUint256)

// Check balance before the swap
const wavaxBalanceBefore = await wavaxTokenContract.balanceOf(account1.address);
const pngBalanceBefore = await pngTokenContract.balanceOf(account1.address)
expect(wavaxBalanceBefore).eq("1000000000000000000000000");
expect(pngBalanceBefore).eq(0)

// We call Swapper contract to make a swap from Wavax to Png. I chose some weird values for the swap cause it's just for
the sack of this tutorial.
await swapper.swap(100, [WAVAX_ADDRESS, PNG_ADDRESS], "0xd7538cABBf8605BdE1f4901B47B8D42c61DE0367", 1000000000,
1807909162115)
// Check balance after

const wavaxBalanceAfter = await wavaxTokenContract.balanceOf(account1.address);
const pngBalanceAfter = await pngTokenContract.balanceOf(account1.address)

// Since we have done the swap, we expect the balance to be slightly different now. Less Wavax and more Png.
expect(wavaxBalanceBefore).lt(wavaxBalanceAfter);
expect(pngBalanceBefore).gt(pngBalanceAfter);
});

});

```

First we have all the imports.

```

import { ethers } from "hardhat";
import * as dotenv from "dotenv";
import { SignerWithAddress } from "hardhat-deploy-ethers/src/signers";
import { BigNumber } from "ethers";
import { Swapper, IWAVAX } from "../typechain";

dotenv.config();

```

I won't go over in details about those, just notice that we use `typechain`, which is a tool that generate automatically typescript bindings for your solidity contracts. Basically it means that, when we instantiate an object corresponding to a Solidity contract, we will have full typings and auto-completion. It saves you a lot of time and help you write better and safer code. (I can't emphasize enough how much I love Typescript)

In the snippet below we can see the `beforeEach` function (it is a [hook](#) actually) that will run before each test case we write in this file.

```

describe("Swappity swap", function () {

 let swapper: Swapper;
 let account1: SignerWithAddress;

 beforeEach(async function () {
 await ethers.provider.send(
 "hardhat_reset",
 [
 {
 forking: {
 jsonRpcUrl: AVALANCHE_NODE_URL,
 blockNumber: 2975762,
 },
 },
],
);

 let accounts = await ethers.getSigners()

 account1 = accounts[0]

 // Here we get the factory for our Swapper contract and we deploy it on the forked network
 const swapperFactory = await ethers.getContractFactory("Swapper")
 swapper = await swapperFactory.deploy(process.env.WAVAX_ADDRESS as string, "0xE54Ca86531e17Ef3616d22Ca28b0D458b6C89106");
 });
 ...
}

```

Couple of things to note here :

- `await ethers.provider.send("hardhat_reset", ...)` It will reset the state of your C-Chain fork. Meaning that each one of your test will run on a clean instance.
- `let accounts = await ethers.getSigners()` Ethers provides us a way to get access to some ``Signers''. Which is a way to represent C-Chain account that we can use in our tests.
- `const swapperFactory = await ethers.getContractFactory("Swapper")` We get here via Ethers a ContractFactory that is an abstraction used to deploy smart contracts.

- `swapper = await swapperFactory.deploy(process.env.WAVAX_ADDRESS as string, "0x...");` Here we use the factory to actually deploy the contract on the hardhat network, which is a forked version of C-Chain Mainnet ! The resulting `swapper` is an object (fully typed thanks to typechain) that represent the `Swapper` contract, and on which you will be able to call functions, like the `swap` one !

So here we see the code of a test. We will break it down a bit more and explain each important portion below.

```
it("should swap wavax for png", async function () {
 // We get an instance of the wavax contract
 const wavaxTokenContract = await ethers.getContractAt("IWAVAX", WAVAX_ADDRESS)
 // @ts-ignore
 const pngTokenContract = await ethers.getContractAt("IWAVAX", PNG_ADDRESS)
 //makes sure owner has enough WAVAX balance
 if ((await wavaxTokenContract.balanceOf(account1.address)).lt("10000000000000000000000000000000")) {
 await wavaxTokenContract.deposit({
 value: BigNumber.from("10000000000000000000000000000000")
 .sub(await wavaxTokenContract.balanceOf(account1.address))
 })
 }

 // We tell Wavax contract that we are cool with Swapper contract using our Wavax on our behalve
 await wavaxTokenContract.approve(swapper.address, ethers.constants.MaxUint256);

 // Check balance before the swap
 const wavaxBalanceBefore = await wavaxTokenContract.balanceOf(account1.address);
 const pngBalanceBefore = await pngTokenContract.balanceOf(account1.address);

 expect(wavaxBalanceBefore).eq("1000000000000000000000000");
 expect(pngBalanceBefore).eq(0);

 // We call Swapper contract to make a swap from Wavax to Png. I chose some weird values for the swap cause it's just for the
 // sake of this tutorial.
 await swapper.swap(100, [WAVAX_ADDRESS, PNG_ADDRESS], "0xd7538cABBF8605BdE1f4901B47B8D42c61DE0367", 100000000,
 1807909162115);

 // Check balance after
 const wavaxBalanceAfter = await wavaxTokenContract.balanceOf(account1.address);
 const pngBalanceAfter = await pngTokenContract.balanceOf(account1.address);

 expect(wavaxBalanceBefore).lt(wavaxBalanceAfter);
 expect(pngBalanceBefore).gt(pngBalanceAfter);
});
```

In the following snippet we see how we can get an instance of a contract at a specific address. So here what we are doing is asking Ethers to give us an object that is a reference to a contract deployed at `WAVAX_ADDRESS` and `PNG_ADDRESS`, with `IWAVAX` as ABI.

Then we check the balance of the account we will use, and if the balance is too small for our taste, we deposit a bit of AVAX in it.

```
// We get an instance of the wavax contract
const wavaxTokenContract = await ethers.getContractAt("IWAVAX", WAVAX_ADDRESS)
// @ts-ignore
const pngTokenContract = await ethers.getContractAt("IWAVAX", PNG_ADDRESS)
//makes sure owner has enough WAVAX balance
if ((await wavaxTokenContract.balanceOf(account1.address)).lt("10000000000000000000000000000000")) {
 await wavaxTokenContract.deposit({
 value: BigNumber.from("10000000000000000000000000000000")
 .sub(await wavaxTokenContract.balanceOf(account1.address))
 })
}
```

Now we tackle the interesting bit of this test, the actual call to our `Swapper` contract.

We can see that we interact with contract we did not deploy, thanks again to the fork feature. For example here we call the approve function of the WAVAX contract. And we also check the balance of our address before the swap.

Then we do the actual call to the swap function of Swapper contract. Passing in the necessary parameters. Here is an overview of the parameters :

- `100` , the minimum amount of token we want to receive for this swap. It's intentionally put low as it is just a test.
- `[WAVAX_ADDRESS, PNG_ADDRESS]` an array of address that correspond to the path we want to take, basically saying that we want to go from WAVAX to PNG token.
- `0xd7538cABBF8605BdE1f4901B47B8D42c61DE0367` correspond to the address of the Pangolin Liquidity Pair for WAVAX \* PNG.
- `100000000` the amount of wavax we are ready to swap for PNG.
- `1807909162115` this can be ignored as it correspond to the deadline parameter. Which is useless on avalanche because transactions are finalized in a very short time frame (< second most of the time).

Then we fetch again the balance of our address. And we check if the balances correspond to our assumptions. If it does, it means that our code working as we expect for this functionality.

```

// We tell Wavax contract that we are cool with Swapper contract using our Wavax on our behalve
await wavaxTokenContract.approve(swapper.address, ethers.constants.MaxUint256);

// Check balance before the swap
const wavaxBalanceBefore = await wavaxTokenContract.balanceOf(account1.address);
const pngBalanceBefore = await pngTokenContract.balanceOf(account1.address);

expect(wavaxBalanceBefore).eq("10000000000000000000000000000");
expect(pngBalanceBefore).eq(0);

// We call Swapper contract to make a swap from Wavax to Png. I chose some weird values for the swap cause it's just for the
// sack of this tutorial.
await swapper.swap(100, [WAVAX_ADDRESS, PNG_ADDRESS], "0xd7538cABBf8605BdE1f4901B47B8D42c61DE0367", 1000000000, 1807909162115);

// Check balance after
const wavaxBalanceAfter = await wavaxTokenContract.balanceOf(account1.address);
const pngBalanceAfter = await pngTokenContract.balanceOf(account1.address);

expect(wavaxBalanceAfter).lt(wavaxBalanceBefore);
expect(pngBalanceAfter).gt(pngBalanceBefore);

```

If you want to see the code in action, you should run this command in the terminal : `npx hardhat test`

This should produce an output looking like this:

Yay ! We successfully tested our contract using a fork of the Avalanche's C-Chain Mainnet.

## Bonus

### Time Travel

Now I still have a couple of things to introduce to you.

Let's say you want to test a compounder contract that work on the WAVAX/PNG pair. What you are most interested in is to see if your compounder contract can reinvest the farm reward into the farm. Issue is that this reward is `time bound`, meaning that you need to wait a bit to see your reward going up. Hardhat provides a way to easily test this sort of situation.

In the snippet below you see that we call a function from the `HardhatRuntimeEnvironment` that will change the time. Then we mine a new block. This will allow you to 'artificially' get a week worth of reward from a pangolin farm and you should be able to test your compounder contract ! Awesome isn't it ?

```

// Advance the time to 1 week so we get some reward
await hre.ethers.provider.send('evm_increaseTime', [7 * 24 * 60 * 60]);
await network.provider.send("evm_mine");

```

### Impersonation

There is another Hardhat's feature that is quite useful: the `impersonation`. With this feature, you can invoke contract call as if you were someone else, like the owner of a contract that is already deployed for example.

In the snippet below we want to call the function `setCoverageAmount` from the `elkIlpStrategyV5`. Which is only callable by the owner of the contract. So not by an address we have the control of. Look at the following snippet.

```

// We impersonate the 'owner' of the WAVAX-ELP StakingRewardsILP contract
await ethers.provider.send('hardhat_impersonateAccount', ['0xcOffexxxxxxxxxxxxxxxxxxxxxx']);
const admin = await ethers.provider.getSigner('0xcOffexxxxxxxxxxxxxxxxxxxxxx')

const stakingcontract = await ethers.getContractAt('IStakingRewardsILPV2', elpStakingRewardAddress, admin);
// We set the coverage elpBalanceAccount1 for our Strategy
await stakingcontract.setCoverageAmount(elkIlpStrategyV5.address, 1000000000000);

await hre.network.provider.request({
 method: "hardhat_stopImpersonatingAccount",
 params: ["0xba49776326A1ca54EB4F406C94Ae4e1ebE458E19"],
});

```

You see here that we start with

```

await ethers.provider.send('hardhat_impersonateAccount', ['0xcOffexxxxxxxxxxxxxxxxxxxxxx']);
const owner = await ethers.provider.getSigner('0xcOffexxxxxxxxxxxxxxxxxxxxxx')

```

Meaning that we will `impersonate` the address `0xcOffexxxxxxxxxxxxxxxxxxxxxx` which is the `owner` of the `IStakingRewardsILPV2` contract.

We can then use the `admin` signer to interact with the contract, as we see in the following section:

```
const stakingcontract = await ethers.getContractAt('IStakingRewardsILPV2', elpStakingRewardAddress, owner);
```

## Conclusion

In this tutorial learned how to set up our Hardhat environment to use a fork of avalanche's C-Chain and use it as a base for our tests, If you want to learn more about Hardhat, I can't recommend you enough to have a look at their [official documentation](#)

I hope you learned something with this tutorial, let me know if you spot a mistake, typo ... Also if you would like to have another tutorial on how to use X with Avalanche, let me know !

## Additional Links

If you want to know more about avalanche, here's a bunch of links for you:

[Website](#) | [White Papers](#) | [Twitter](#) | [Discord](#) | [GitHub](#) | [Documentation](#) | [Forum](#) | [Avalanche-X](#) | [Telegram](#) | [Facebook](#) | [LinkedIn](#) | [Reddit](#) | [YouTube](#)

## How to Mint Your Own ERC721 on Avalanche Using Open Zeppelin

This tutorial teaches you to deploy your own ERC721 token on Avalanche Fuji C-chain testnet.

The steps to deploy on the Mainnet are identical and the differences are mentioned in the tutorial.

## Contents

- [Getting images ready to be uploaded to decentralized storage.](#) (IPFS in this tutorial)
- [Getting metadata ready to be uploaded to decentralized storage.](#) (IPFS in this tutorial)
- [Writing code in Remix IDE](#)
- [Compiling and Deploying to Avalanche Fuji C-Chain Testnet.](#)
- [Check the NFT on the explorer.](#)

### Getting Images Ready to Be Uploaded to Decentralized Storage

In order to create NFT we first need to have the image/video content hosted on a decentralized storage solution like IPFS. IPFS in itself won't be enough because if you host it on IPFS and garbage collection takes place your assets will be gone and the NFT will not show your image/video.

For this we need a pinning service like Pinata. So you can make an account on Pinata [here](#).

Now, let's get the images ready. You might have seen on marketplaces like OpenSea NFT's are in a collection like BAYC, CryptoPunks etc.

So do we store links to the metadata of every NFT in the collection? The answer is no.

OpenZeppelin has come up with a smart way where you just need to store the prefix of the url where the metadata is stored and the tokenId is appended to it to return the url of the metadata for an individual NFT. For example, if the baseURI is "<https://mynft.com/>" and if you want the metadata for tokenId 1 the contract simply returns ("<https://mynft.com/1>"). This way we don't need to store the url of the metadata for every tokenId.

To achieve this we need to rename our images/videos with respect to tokenId. Make sure to have all the assets in a single folder if you plan to create a collection of NFT's.

Image / Video for tokenId 0 should be named 0 followed by the extension (jpg, png, tiff, gif, mp4, etc...)



Here I have a single image named 0 to represent the image corresponding to the tokenId 0. You can name your assets similarly starting from 0 or 1 make sure that your contract also start the tokenId from 0 if you start naming from 0 otherwise 1 whichever is your case. In this tutorial, we will start from 0.

Let's upload these assets to pinata for pinning on IPFS.

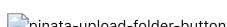
In the pinata dashboard you should see an upload button.



We are supposed to upload the entire folder the reason for that is we get the same baseURI and tokenId type URL if we upload the folder more about it below.



Upload the folder by clicking the "Click to upload" button we don't want a custom name for the pin and we need not preserve the directory name so we can ignore those options for now.



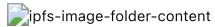
You can see the files that are about to upload since I have only one you can see it in the image below. Click the "Upload" button to start uploading.



Once done you should see the folder in the files section as shown in the image below.



You can now click on the name of the folder to see its content on IPFS.



So now we have the baseURI for the assets to get any asset all we need to do is append the tokenID and extension to the baseURI.

This is the baseURI in my case. <https://gateway.pinata.cloud/ipfs/QmaHGo7pQ9x7B1rNvPbkzTnrZNuHA4mx53t8ZnAA8JFUG2>

To get asset for tokenID 0 all I need to do is append "/" + tokenID + extension (.gif in this case)

<https://gateway.pinata.cloud/ipfs/QmaHGo7pQ9x7B1rNvPbkzTnrZNuHA4mx53t8ZnAA8JFUG2/0.gif>

### Getting Metadata Ready to Be Uploaded to Decentralized Storage

Now that we have the baseURI for assets we need to prepare the metadata that the marketplaces parse in order to extract attributes and artwork from it.

Similar naming convention has to be followed for metadata also in case you plan to create an NFT collection.

The content of metadata files is expected to be in .json format. However, files need not have an extension. So the metadata for tokenID 0 would be `0.json` however when we upload to pinata we suppress the extension.

Create a separate folder to store all the metadata files. Create a text file using a notepad, notepad++, or text editor of your choice.

This is the metadata format expected by marketplaces like OpenSea.



- Let's go through every attribute one by one.
  - `name` - specify the name of the NFT.
  - `tokenId` - specify the tokenID of the NFT.
  - `image` - specify the URL where the assets for the NFT are hosted. This is the same URL I have given above. Make sure to include the complete URL including the tokenID and extension part.
  - `description` - specify some description about the entire NFT collection.
  - `attributes` - specify the attributes of the NFT. Notice the format to specify the attributes.

I have also attached the same metadata file in the repository in case you want to copy and edit it go for it.

You will need to create such a metadata file for every NFT in the collection. Usually, people write a script to generate metadata files.

Once you are done you should have a folder of metadata files ready to be uploaded to pinata.



Upload the folder of metadata to pinata.

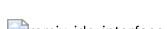


You can click on the folder name and see the contents on IPFS.

### Writing Code in Remix IDE

Let's now write the code for the ERC721 token. In this tutorial we will use the [Remix IDE](#) for deploying smart contracts as it is beginner-friendly.

This is how the Remix IDE interface looks like and we can see our project files on the left.



Let's create a file under the `contracts` folder for our token.



For this tutorial I named it `MyNFT.sol` make sure to have the `.sol` extension. As a good practice, you can name the file with the same name as the token.



I have attached the code for the NFT in the same repository.

- Let's go through the code.
  - Inherits
    - `ERC721("MyNFT", "MN")` - replace `MyNFT` with the name you want the collection to have, replace `MN` with the symbol you want for the collection.
    - `Ownable` - ownable is used to have an access control mechanism. Ownable provides utility functions to `getOwner`, `setOwner` and `renounceOwnership`.
  - Variables

- `tokenCounter` - It is used to keep track of the tokenId to mint.
  - `price` - specify the price you want the minter to pay (in Wei) (0.01 AVAX in this case).
  - `MAX_TOKENS` - specify the maximum number of NFT that can be minted from this collection. (100 in this case)
  - `baseURI` - the base URL for the metadata stored on Pinata. (*you will need to specify this*)
- Functions
    - `mint` - the main mint function that is to be called in order to create an NFT. It consists of 2 require statements one to check if the max supply is exceeded and the second to check if the minter is paying the correct price.
    - `_baseURI` - we need to override the default OpenZeppelin `_baseURI` because the default one returns an empty string.
    - `withdraw` - this function can be called by the owner of the NFT collection to withdraw the funds deposited by the NFT minters. 😊

### Compiling and Deploying to Avalanche Fuji C-Chain Testnet

Let's now compile the code to check if there are any errors. To compile the shortcut is Ctrl / Command + S or click the icon shown in the image below.



Make sure you have the correct compiler selected from the dropdown and click the "Compile" button.



Once compiled we can now deploy the contract to the Avalanche Fuji C-Chain testnet. *The steps to deploy on the Mainnet are the same.*

Click the deploy button as shown in the image below.



In the deploy section make sure you have "Injected web3" selected in the dropdown.



Make sure it shows the correct account that you want to use to deploy the NFT. Make sure the correct contract is selected to be deployed.

Once ready you can hit the "Deploy" button.

You should get a "Confirm Transaction" prompt, hit the "Confirm" button.



You should get a MetaMask pop-up asking to confirm the transaction.



Make sure you are on the correct network and hit the "Confirm" button. You should be able to see the contract deployment under progress in your MetaMask.

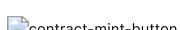


Once the contract is deployed you should be able to see it under the "Deployed Contracts" section.



Nice! We now have deployed the contract however we don't own any NFT. To own an NFT we need to mint it.

Expand the contract and see for the "mint" button.



You won't be able to mint unless you pay 0.01 AVAX in order to mint. For that, you will need to specify the amount to pay. Remix IDE doesn't let you specify decimals so we need to specify it in finney which is a lower unit than ether. 1 ether = 1000 finney. We need 0.01 ether so 10 finney.



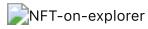
Now try clicking the "mint" button. "Confirm Transaction" dialog might appear confirm it. Make sure you are on the correct network, the amount is correct, and hit the "Confirm" button.

Hooray! We have now minted an NFT to ourselves! 🎉

### Check the NFT on the explorer

You can check the NFT on the explorer. My URL is below.

<https://testnet.snowtrace.io/tokens/0x10075f07b799f9ce7a585e95a2711766b1e248a2/instance/0/token-transfers>



The format is as follows.

Testnet - <https://testnet.snowtrace.io/tokens/{contract-address}/instance/{tokenId}/token-transfers>

Mainnet - <https://snowtrace.io/tokens/{contract-address}/instance/{tokenId}/token-transfers>

## Conclusion

We have successfully minted an ERC721 on the AVAX blockchain. OpenZeppelin is doing a pretty great job at providing boilerplate code to get started however there is a lot more things you can do to an NFT. Make the price go up on every mint 😊 or making the NFT artwork dynamic by serving the images from a server instead of IPFS 😊. You may also use alternate decentralized storage solutions like Arweave.

I recommend diving more deep and understand every function available to override (basically editing it your way). You can learn more about it [here](#) and [here!](#)

Happy Minting! 😊

# How to Use The Graph to Query Avalanche Data

## Table of Content

- [Introduction](#)
- [What is The Graph](#)
- [The Graph and Open APIs](#)
- [GraphQL Introduction](#)
- [What is Avalanche](#)
- [The Graph and Avalanche](#)
- [Interacting with Avalanche Data via The Graph: Pangolin Example](#)
- [How to Build a Subgraph](#)
- [How to Deploy a Subgraph](#)
- [Conclusion](#)

## Introduction

The importance of data is nowadays encapsulated by the saying, "data is the new oil". What this means is that properly harnessed data is now an invaluable part of any economic strategy, information gathering, or decision-making process as insights gleaned from data can open up productive opportunities and serve as a competitive advantage. The new economy that is being created by the innovation of [blockchain](#) technology is no exception. Blockchain data is notoriously difficult to index and query as there isn't an inbuilt query language as with traditional databases. Thankfully, breakthroughs and new approaches are being made in this space to facilitate data accessibility. One of such protocols that seek to simplify the handling of blockchain data is [The Graph](#). In this tutorial, you will be taught how to query data from [Avalanche](#) blockchain using The Graph. Avalanche is a next-generation blockchain that lays emphasis on being incredibly fast, low-cost, and scalable. It is also eco-friendly as it relies on a novel approach to consensus, more on this later. The layout of this tutorial is divided into three broad logical sections. First, you will be introduced to The Graph protocol, its mission of creating open APIs standards, and the underlying query language it utilizes, [GraphQL](#). Second, you will be given an overview of Avalanche, how it differs from competitor blockchains, and its design decisions that aid developer onboarding and satisfaction. Third, you will be shown how to tie the concepts learned from The Graph and Avalanche to develop a sound data strategy that enables efficient querying of Avalanche data in your Avalanche applications. By the end of this tutorial, you will be well vested with the knowledge required to create your own data-driven Avalanche applications.

## What Is the Graph

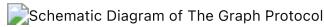
The Graph is an open-source protocol for indexing and querying blockchain data. The main value proposition of The Graph is to enable anyone to create open APIs that power a decentralized future. Unlike traditional APIs in the Web2 world which are often centralized, proprietary, and have gatekeepers that restrict access, The Graph envisions an open ecosystem of APIs where one API can be built on top of another without needing permission. This can be achieved through the usage of [subgraphs](#) which can be thought of as APIs assembled through The Graph that index specific data and exposes querying capabilities to other users. The network of various subgraphs exposes a global graph that provides access to public information in an open and transparent way.

The Graph is currently made up of two types of services, a [hosted service](#), and a [decentralized Graph network](#). The hosted service is free to use and supports a wide range of blockchains like Avalanche, Ethereum, Fantom, Polygon, Binance Smart Chain (BSC), etc. alongside several testnets of the aforementioned chains. The decentralized Graph network on the other hand currently only supports Ethereum, however, the long-term vision is to have it support other chains as in the hosted service. The main difference between both is that the hosted service is a centralized offering where the nodes are run and maintained by The Graph's own team, whereas the Graph network is a decentralized offering in which members of the public can operate nodes that support and guarantee the security of the network in a permissionless manner. The token of The Graph decentralized network is GRT (The Graph) and various actors in the ecosystem are incentivized through token inflation to bootstrap the network. Developers can create subgraphs and have indexers run nodes that index those subgraphs, just as curators can signal or support reliable subgraphs through their tokens while delegators who may be non-technical can still contribute to the network by delegating their token to indexers (those who run nodes). The decentralized network is thus more censorship-resistant than the hosted service and as it matures in production, it will eventually support other blockchains apart from Ethereum and eclipse the hosted service which came first.

If you look under the hood of The Graph protocol, you will discover that the hosted service and the decentralized Graph network both rely on an open-source implementation of a [Graph Node](#). The Graph Node is a software that can be run which deterministically stores events triggered from blockchains such as Ethereum in a data store. For the initial implementation, the data store used was [PostgreSQL](#), and a Graph Node is usually expected to be run alongside an [IPFS](#) node and a full node capable of validating all data from that blockchain, for example, an Ethereum full node. With the hosted service, you do not need to run these nodes as they are managed for you. As a result, the hosted service is a great place to start using The Graph and all examples in this tutorial will use the hosted service.

## The Graph and Open APIs

In this section of the tutorial, you will be shown the integral pieces of The Graph protocol, how they fit together and what a typical workflow looks like.



Source: <https://github.com/graphprotocol/graph-node/blob/master/docs/getting-started.md>

The schematic diagram above is an overview of a dapp using The Graph protocol and it contains the components and typical workflow involved. At the center is the Graph Node which was talked about in the last section, it listens to events triggered by smart contracts on the blockchain that it indexes and records those events as data in a store through a predefined mapping (the [WASM](#) module) that transforms that data into the desired format. The Graph Node exposes a GraphQL API that the dapp can use to query data from the blockchain it wants to display in the frontend. When such queries are received, Graph Node retrieves the relevant data from the attached store and serves the requests. Note that transactions in the dapp are initiated directly on the smart contracts on the blockchain and do not pass through Graph Node as there is no write capability. The Graph protocol is used to index and query historic data so it can be seen as having a read-only mechanism. However, as a result of its architecture, caching, and storage, data is queried efficiently since relevant events were indexed earlier. This is a much better approach than having dapps query the blockchain directly to read historic data.

To better understand the workflow of The Graph protocol, it can be easier to think of it as being made up of three components, the Graph Node, the [Graph CLI](#), and the [Graph TypeScript Library](#). The Graph Node has been explained extensively above, the Graph CLI is a Command Line Interface through which you can interact with The Graph protocol and do things like create a subgraph, unregister a subgraph, generate scripts for smart contracts to be indexed, deploy a subgraph to the Graph Node, etc. The Graph TypeScript Library provides a set of helper APIs for you to access the underlying store, blockchain data, smart contracts, IPFS files, cryptographic functions, etc. It can be thought of as providing a connector/mapping from one domain to the other.

The entry point of a subgraph project is the manifest file. It contains the general definitions of the subgraph such as the smart contracts being indexed, the names of events of interest, the handler functions that are triggered on those events, etc. Other important files that make up a subgraph project are the GraphQL schema file, the mappings file that contains code that dictates how entities are accessed and stored, and a bunch of other auto-generated files which you will understand in-depth in subsequent sections. For now, the most important thing to understand is that the Graph protocol exposes blockchain data via GraphQL APIs, and it does this through an internal mapping of that data to a local store.

## GraphQL Introduction

GraphQL is a query language for APIs and it solves some of the pain points associated with traditional APIs that use the [REST](#) framework. With GraphQL, data can be gotten from different resources in a single request as there is no need for multiple round trips to different endpoints to fetch data. GraphQL also enables frontend clients to describe and request exactly the data they are interested in unlike REST APIs where data that may not be required by the client is returned simply because it is a part of that endpoint. GraphQL uses a single endpoint to serve requests as it relies on the concept of types and fields so your APIs can evolve without the need for explicit versioning, rather new fields, and types can be added. This decouples API design from the frontend structure as APIs can be developed independently and frontends can ask for only data they are interested in with guarantees that they will receive that data if they follow the schema provided by the GraphQL API.

GraphQL defines three [operation types](#) namely Query, Mutation, and Subscription. In this tutorial, you will concentrate on the [features of GraphQL as relates to The Graph](#). The Graph only supports the Query type, which as the name implies is used for querying (retrieving) data. Mutations represent actions that can change or update data and are not supported as dapp developers are expected to interact directly with the underlying blockchain through transactions. Subscriptions are used for maintaining an existing connection from a client to a GraphQL server and are not supported by The Graph.

In GraphQL, the Query type is the entry point to the API, The Graph defines an Entity type for its schema but automatically creates the Query type as the top-level type (root type) for you with fields - entity and entities. Below is an example.

```
type Token @entity {
 id: ID!
 address: Bytes!
 name: String!
 symbol: String
}
```

The entity in this example is Token, the `@entity` directive is provided by The Graph and will create fields token and tokens in the automatically generated Query type. Note that these fields generated are what will be used in constructing a GraphQL query as you will see shortly. The fields in the Token entity - id, address, name, symbol are the data that will be returned if they are included as part of a query. An entity must also include an id field for it to be considered valid. The exclamation mark indicates that the field is NON NULLABLE, which means it will always have a value.

By looking at the GraphQL schema above, you can construct a query that closely mirrors it and includes only the data you are interested in. An example query is shown below.

```
{
 token(id: "1") {
 id
 address
 }
}
```

The example query is for a specific token entity with an `id` of `1`. When querying for a single entity in The Graph, the `id` field must be included. The result returned will mirror the query structure and will only include data requested for which in this case is the `id` and `address` fields. A probable result is shown below.

```
{
 "data": {
 "token": {
 "id": "0x10",
 "address": "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2"
 }
}
```

```
}
```

You can also query for all tokens (entities) using the query below.

```
{
 tokens {
 id
 address
 name
 symbol
 }
}
```

The results will match the query structure and will be an array of tokens with the additional fields - `name` and `symbol` included because they were requested for unlike in the last query example.

```
{
 "data": {
 "tokens": [
 {
 "id": "0x10",
 "address": "0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2",
 "name": "Wrapped Ether",
 "symbol": "WETH"
 },
 {
 "id": "0x11",
 "address": "0x2260fac5e5542a773aa44fbcedf7c193bc2c599",
 "name": "Wrapped BTC",
 "symbol": "WBTC"
 }
]
 }
}
```

The Graph also includes support for sorting, pagination, filtering, time-travel queries, etc. please refer to [The Graph documentation](#) for a full list of features.

## What Is Avalanche

Avalanche can best be described as a blockchain ecosystem and smart contracts platform that is built from the ground up to combat the notion that blockchains are inherently slow and not scalable. The architecture of Avalanche is comprised of [3 main chains](#) which derive their security guarantees from the [primary network](#). The chains are the [Exchange Chain \(X-Chain\)](#), the [Platform Chain \(P-Chain\)](#), and the [Contract Chain \(C-Chain\)](#). The Exchange Chain is used for creating and trading digital assets and relies on Avalanche Consensus Protocol, the Platform Chain is used for creating new [Subnets](#) and custom blockchains. The Contract Chain is an instance of the [Ethereum Virtual Machine \(EVM\)](#) running on Avalanche. Avalanche, therefore supports Ethereum tooling and [Solidity](#) as smart contracts built for Ethereum can be deployed on the Contract Chain with the added benefit of increased throughput. Decentralized applications can also be built natively for the Contract Chain using tools like [MetaMask](#), [Truffle](#), [Waffle](#), [Remix](#), etc. For more explanations about concepts in Avalanche, you can consult the official [documentation](#).

Below is a diagram that further illustrates the points discussed.



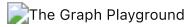
Source: <https://docs.avax.network/learn/platform-overview>

## The Graph and Avalanche

In line with The Graph's vision of providing open access to the world's data and Avalanche's goal of being an open, programmable, smart contracts platform that powers the next wave of scalable decentralized applications, both entities [announced a partnership](#) that allows on-chain data from Avalanche to be indexed and queried through The Graph protocol. This means that Web3 developers can now access Avalanche data through subgraphs and build data-intensive applications while also having access to data from other blockchains supported by The Graph protocol.

## Interacting with Avalanche Data via the Graph: Pangolin Example

In this section, you will learn how to query Avalanche data from an already deployed subgraph on The Graph's [hosted service](#). The subgraph you will use indexes data from the [Pangolin](#) decentralized exchange on Avalanche. Pangolin is an [Automated Market Maker \(AMM\)](#) on Avalanche, similar in operation to [Uniswap](#). It can be used to swap Ethereum and Avalanche assets and has fast settlement times and low transaction fees. The Graph's hosted service provides a playground for deployed subgraphs. The playground is an IDE-like environment based on [GraphQL](#) where you can write sample queries to fetch data supported by a subgraph. It is a valuable tool when getting to know the queries that are supported by third-party subgraphs deployed by various projects. The Pangolin exchange subgraph can be accessed [here](#). Below is an image of the playground interface.



Source: <https://thegraph.com/legacy-explorer/subgraph/dasconnor/pangolin-dex?selected=playground>

The playground consists of the 3 sections, the query section, the results section, and the schema section. You can modify queries, click on the play button and have the returned data displayed in the middle section (results section). The schema section enables you to go through the entities and their associated fields to know what kind of queries are supported by that particular subgraph. Below is an example query you can initiate on the Pangolin exchange subgraph.

```
{
 tokens(first: 2) {
 id
 symbol
 name
 decimals
 }
}
```

The query above is asking for the first 2 tokens from the results of the Pangolin exchange subgraph and it is interested in the fields - `id` , `symbol` , `name` , and `decimals` . The returned result is shown below.

```
{
 "data": {
 "tokens": [
 {
 "decimals": "18",
 "id": "0x008e26068b3eb40b443d3ea88c1ff99b789c10f7",
 "name": "Zero.Exchange Token",
 "symbol": "ZERO"
 },
 {
 "decimals": "18",
 "id": "0x020ef96c76a225fc0151c191b2a15accc915b68c",
 "name": "WOUF",
 "symbol": "WOUF"
 }
]
 }
}
```

Using the schema, you can browse through the Pangolin API to discover other entities you are interested in. For example, you can construct a query to know the daily volume on Pangolin and the number of transactions carried out on the [DEX](#).

```
{
 pangolinDayDatas(first: 3, orderBy: date) {
 date
 txCount
 dailyVolumeETH
 dailyVolumeUSD
 }
}
```

The query above will request the data from the first 3 days and only the fields specified will be returned.

```
{
 "data": {
 "pangolinDayDatas": [
 {
 "dailyVolumeETH": "0",
 "dailyVolumeUSD": "0",
 "date": 1612742400,
 "txCount": "2"
 },
 {
 "dailyVolumeETH": "52995.5502782044446352341877709734",
 "dailyVolumeUSD": "1497153.144890008456466344638128839",
 "date": 1612828800,
 "txCount": "3603"
 },
 {
 "dailyVolumeETH": "381001.9576775585402832922123431514",
 "dailyVolumeUSD": "15686187.55247135303384368066876295",
 "date": 1612915200,
 "txCount": "12447"
 }
]
 }
}
```

The result shows that there was no volume on the first day but the volume and the number of transactions increased steadily from the second day to the third day.

More data can be gotten from this particular entity by constructing queries that match the fields supported. These fields can be identified from the schema as shown in the image below.



Source: <https://thegraph.com/legacy-explorer/subgraph/dasconnor/pangolin-dex?selected=playground>

You can also interact with a deployed subgraph programmatically from your frontend application. To do this, you will need a GraphQL client like [Apollo](#) that will act as a communication layer between your application and The Graph. First, open up a terminal in your project folder and install Apollo and GraphQL. You can use either [NPM](#) or [Yarn](#) to install both using the commands below so.

```
npm install @apollo/client graphql
yarn add @apollo/client graphql
```

Then get the API URL from the subgraph you want to use. The API URL can be gotten from The Graph explorer page of the deployed subgraph. In the Pangolin example which you have been working on, the API URL for HTTP queries is <https://api.thegraph.com/subgraphs/name/dasconnor/pangolin-dex>.

Import Apollo client to your project's code, construct a query according to the schema of the subgraph you are querying and issue the query using your instantiated Apollo client. A code snippet demonstrating this is shown below.

```
import { ApolloClient, InMemoryCache, gql } from '@apollo/client';

const APIURL = "https://api.thegraph.com/subgraphs/name/dasconnor/pangolin-dex";

const tokensQuery = `
query {
 tokens (first: 5) {
 id
 symbol
 name
 decimals
 }
}

const client = new ApolloClient({
 uri: APIURL,
 cache: new InMemoryCache()
});

client.query({
 query: gql(tokensQuery)
})
.then(data => console.log("Subgraph data: ", data))
.catch(err => { console.log("Error fetching data: ", err) });
```

For a full reference on how to use Apollo Client, kindly consult the official [documentation](#).

## How to Build a Subgraph

In the previous section, you were shown how to integrate an already deployed subgraph into your project. In the next two sections, you will go a step further to define your subgraph, build it and deploy it to the hosted service.

Follow the instructions below to begin the process.

- The first thing to do is to sign in to the [hosted service](#) if you already have an account. If you do not have an account, you will be required to create one using your [GitHub](#) profile.
- Next, click on your GitHub profile picture at the top right-hand corner to access the Dashboard menu item.
- Once on the Dashboard click on the Add Subgraph button.
- Next, on the Create a Subgraph page, input relevant details like your subgraph name, subtitle, description, GitHub URL, etc. then create the subgraph.
- You will be presented with a page containing instructions on the next steps. Do not worry about these as you will perform them in this section.



The subgraph project you will build will be based on the governance token of the Pangolin exchange [PNG](#). It will be a simple project with a single entity - `Transfer`, which will be used to track Transfer events of the token. At the end of building this project, you will have hands-on experience of how the various pieces fit together in a Graph project.

First, you need to install Graph CLI as you will use it to interact with The Graph from your terminal. Install it using NPM or Yarn with the commands below.

```
npm install -g @graphprotocol/graph-cli
yarn global add @graphprotocol/graph-cli
```

Next, you will initialize a scaffold project using Graph CLI like so.

```
graph init --product hosted-service <GITHUB_USER>/<SUBGRAPH NAME>
```

Replace the variables in the command above with your GitHub username and the name of the subgraph you created in the earlier steps. Note that if your subgraph name contains spaces, it will be concatenated with dashes and will be in lowercase. So `Pangolin Token` will become `pangolin-token`.

You will be prompted in the terminal to answer some questions such as the subgraph name, the directory to create the subgraph, the blockchain network, etc. Do not forget to switch the network to Avalanche.



In a typical workflow where you are building your own smart contracts for Avalanche Contract Chain, you will supply your deployed smart contract address when prompted to do so in the terminal. The smart contract serves as the data source for your subgraph. In our example, we will use the Pangolin token which is already deployed on Avalanche. Do note that if The Graph fails to automatically detect your deployed smart contract address from the online block explorer, you will be required to provide a path to the contract's Application Binary Interface (ABI) file. If it is a project you are building you can generate the ABI files from your Solidity code. If it is an external project, you can download the source code from the project's GitHub repository and generate the ABI files locally. Alternatively, you can copy the ABI of the deployed contract from [Avalanche Contract Chain Explorer](#) into a file. However, this is not the recommended approach as the version you find on the Contract Chain Explorer may not be the latest, also you run the risk of erroneously copying a malicious ABI. If you must copy the ABI from the explorer, make sure you get the contract address from the project's documentation. For Pangolin token, you can find it [here](#). Copy the Pangolin token ABI from the Contract Chain Explorer [code tab](#), save it in a file with the name `png.json` and provide the file path to the Graph CLI when asked if you run into issues with automatic detection from the contract address. Provide the value `png` for the contract name when prompted by the CLI. The Graph CLI will generate a scaffolding containing several files and the supplied ABI. Below is what the generated file structure looks like.



The `subgraph.yaml` file is the main entry point to your subgraph project, it contains the subgraph manifest.

```
specVersion: 0.0.2
schema:
 file: ./schema.graphql
dataSources:
 - kind: ethereum/contract
 name: Png
 network: avalanche
 source:
 address: "0x60781C2586D68229fde47564546784ab3fACA982"
 abi: Png
 mapping:
 kind: ethereum/events
 apiVersion: 0.0.4
 language: wasm/assemblyscript
 entities:
 - Transfer
 abis:
 - name: Png
 file: ./abis/Png.json
 eventHandlers:
 - event: Transfer(indexed address, indexed address, uint256)
 handler: handleTransfer
file: ./src/mapping.ts
```

The `dataSources` field on the `subgraph.yaml` file specifies the smart contract of interest. The value of the `abi` field under the `source` field must match the value of the `name` field under `abis`. The value under `entities` is the name of the GraphQL entity that you will create. The `eventHandlers` field currently contains one event and the associated function that will be triggered to handle it. This file was generated from the ABI file for the Pangolin token. Other events exposed by the ABI have been removed for simplicity. The file `schema.graphql` contains The Graph entities and is referenced in the `subgraph.yaml` file. Below is the content of the file.

```
type Transfer @entity {
 id: ID!
 from: Bytes! # address
 to: Bytes! # address
 amount: BigInt!
}
```

The content contains one entity - `Transfer` indicated by the `@entity` directive. The fields that make up the entity are `id`, `from`, `to`, `address`. Notice that these fields closely mirror the event signature of the `Transfer` event of the Pangolin token. The fields have their type specified and all are NON NULLABLE as indicated by the exclamation mark.

The next important piece is the [AssemblyScript](#) Mappings. AssemblyScript is used to transform data from the blockchain's events and GraphQL entities into a format that can be loaded on The Graph Node. Before writing your event handler mapping for the `Transfer` event in the `mappings.ts` file under the `src` folder, you will need to generate the AssemblyScript classes from your ABI and `schema.graphql` file. Run the command below to do so.

```
graph codegen
```

Whenever you make changes to your GraphQL schema or your subgraph manifest, you will be required to regenerate the mappings using the command above. It is considered best practice to regenerate your mappings often as they can become a source of errors if you forget to do so.

Back in the `mappings.ts` file, you will import the generated classes from the Pangolin token ABI and the GraphQL schema. Next, the `handleTransfer` function will take in a `Transfer` event, parse its contents, and store it in the Graph Node. Remember that a blockchain event is always associated with a handler

function that is triggered whenever the said event occurs. Below is the `mappings.ts` file for the Transfer event.

```
// Import the Transfer event class generated from the Png ABI
import { Transfer as TransferEvent } from "../generated/Png/Png"

// Import the Transfer entity type generated from the GraphQL schema
import { Transfer } from "../generated/schema"

// Transfer event handler
export function handleTransfer(event: TransferEvent): void {
 // Create a Transfer entity, using the hexadecimal string representation
 // of the transaction hash as the entity ID
 let id = event.transaction.hash.toHex()
 let transfer = new Transfer(id)

 // Set properties on the entity, using the event parameters
 transfer.from = event.params.from
 transfer.to = event.params.to
 transfer.amount = event.params.amount

 // Save the entity to the store
 transfer.save()
}
```

You can now build the subgraph project to make sure you do not have any errors. Run the build command like so.

```
graph build
```

If your subgraph builds successfully, you are ready for deployment.

## How to Deploy a Subgraph

To deploy a subgraph you must authenticate with the hosted service using your access token. Your access token can be seen from your project's dashboard. You can authenticate by running the command below.

```
graph auth --product hosted-service <ACCESS_TOKEN>
```

The final step is to run the deployment command below and replace the variables with your details.

```
graph deploy --product hosted-service <GITHUB_USER>/<SUBGRAPH NAME>
```

This initiates the deployment process and you should see your subgraph being uploaded to IPFS. After a while the upload is complete. You can now switch to your project dashboard on the hosted service. Your subgraph will have a status of syncing as it indexes the event of interest from the genesis block of the blockchain. If you want indexing to start at a specific block, you can specify it in the `startBlock` field under the `source` field in your `subgraph.yaml` file. For a full list of options for the subgraph manifest look at the official [documentation](#).



Source: <https://thegraph.com/legacy-explorer/subgraph/ofemeteng/pangolin-token>

Depending on the amount of data that is being indexed by the subgraph, the syncing period may take longer. Once the subgraph has synced fully without errors, you can query the supported entities through the playground or by integrating the queries endpoint in your project using a framework like Apollo.

You can try issuing the query below on your deployed Pangolin Token subgraph from the playground.

```
{
 transfers(first: 3) {
 id
 from
 to
 amount
 }
}
```

The results return the first 3 transfer events from the subgraph as expected.

```
{
 "data": {
 "transfers": [
 {
 "amount": "46649696040618800000",
 "from": "0xeed2db9b2d2645aaca044ecb397518ca6d9e74a5",
 "id": "0x000296c30c325db75de48101737f5d54408af486ab021323225767701429e66e",
 "to": "0xd7538cabbf8605bde1f4901b47b8d42c61de0367"
 },
 {
 "amount": "1766396909474367721",
 "from": "0xbb67987c040619842b0f8b0257bde63be842b27b",
 "id": "0x000296c30c325db75de48101737f5d54408af486ab021323225767701429e66e"
 }
]
 }
}
```

```

 "id": "0x000308522c9a2266f4ac73e13f02e7496decbad08286d52642ee768ff7ef343e",
 "to": "0xd7538cabbf8605bdef4901b47b8d42c61de0367"
 },
 {
 "amount": "6826187345560538842",
 "from": "0xa16381eae6285123c323a665d4d99a6bcfaac307",
 "id": "0x0003aba174d1ad02e24d07122bb8bd66d194b640f79b19f3290885f2952b7b2b",
 "to": "0x063b88d53d109c12ec21785c4e5e89bb71369432"
 }
]
}

```

You can find the full code for the Pangolin Token subgraph in this [GitHub repository](#).

## Conclusion

Congratulations on getting to the end of this tutorial. That was an extensive tour of The Graph and Avalanche ecosystems. In this tutorial, you were introduced to the concepts underpinning The Graph protocol and how it is set to bring about a new era of open APIs through which developers can build on the work of others in an open, decentralized, and permissionless manner. You were also introduced to GraphQL, the query language that powers The Graph protocol. The unique tweaks of The Graph protocol as relates to GraphQL were also highlighted and it was shown that The Graph currently only supports the Query type in the GraphQL specification. Next, you were introduced to the Avalanche blockchain, which was designed specifically to solve some of the scaling issues currently being experienced by other Layer 1 blockchain networks. Avalanche architecture and design choices were highlighted. Even though Avalanche operates based on a novel consensus algorithm, its Contract Chain which is one of 3 chains that make up the Avalanche ecosystem is EVM compatible. This means that developers from Ethereum and other EVM-based chains can use tools that they are already familiar with. Avalanche support for Solidity also means that DApps built for these chains can be deployed on Avalanche and immediately reap the benefits associated with low transfer fees and increased throughput. You were also shown how to use The Graph protocol to query data from Avalanche. The playground provided by The Graph was used to interact with the Pangolin decentralized exchange subgraph and you were also introduced to Apollo Client through which you can programmatically interact with subgraphs via your frontend projects. Finally, you built your own subgraph from scratch and deployed it on the hosted service by leveraging the governance token of Pangolin.

**At this point, you should be well equipped with the knowledge and tools you need to build awesome data-based projects using The Graph and Avalanche. Happy BUIDLing.**

**description:** This tutorial will show you how to setup a Chainlink node with the Avalanche Fuji Testnet and create dapp smart contracts utilizing a Chainlink node.

## How to Configure and Use Your Own Chainlink Node And External Adapter In Your Avalanche dapp

### Introduction

This tutorial will show you how to setup a Chainlink node with the Avalanche Fuji Testnet and create dapp smart contracts to connect to the Chainlink node.

We at [red.dev](#) needed to do learn how to do this for our current software project under development, [RediYeti](#). In our case, we needed our dapp (on the C-Chain) to gather information from the Avalanche P-Chain, and because Avalanche does not allow this natively, we built a Chainlink adapter to do this job.

You, however, can follow this same methodology to gather any real-world information that your dapp needs, by following this tutorial and just designing your Chainlink adapter to gather different information.

In this tutorial, we describe each step of setting up the environment by hand. However, to make your life easier after you learn how it works, we have included a set of Ansible scripts to complete this process automatically. (For more information on the devops tool [Ansible](#), see the [Resources](#) section at the end of this tutorial.) Ansible creates a development server using a Vultr.com vps, and you can find the entire project [here](#).

### Audience

To get the most out of this tutorial, you will need to have a basic understanding of Docker, Chainlink, JavaScript, Node, Solidity, and how to write dapps. If you do not yet know about these topics, see the [Resources](#) section at the end for links to learn more.

### Overview

At the very highest level, here is an overview of the process we will take you through in this tutorial. First we are going to show you how to install the necessary software required to run the Chainlink node, and then we will explain to you how to run the Chainlink node within the Docker container. After that, with the help of the Chainlink GUI, we can create a simple job which will be used in a smart contract to gather real-world data.

### Prerequisites

1. Ubuntu 20.04 or later
2. [Docker-CE](#)
3. [Go 1.16.6](#)
4. [AvalancheGo v1.4.5](#) or higher
5. [MetaMask wallet](#) set up for use with Avalanche Fuji Testnet
6. User with `sudo` access

### Install Docker-CE

The first step is to install Docker-CE. Docker-CE (community edition) is a free version of Docker that you can use to spin up containers without having to pay for enterprise-level support.

Open a terminal session and execute the command below:

```
curl -sSL https://get.docker.com/ | sh
```

To manage Docker as a non-root user, create a docker group and add your user to it.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Verify that you've installed Docker by running the command below:

```
docker -v
```



The next step is to download and install the Go language (a.k.a. "Golang") which is required for building AvalancheGo, later in this section.

## Install Go

Download the Go package. We have used version 1.16.6 for this tutorial:

```
wget https://storage.googleapis.com/golang/go1.16.6.linux-amd64.tar.gz
```

Extract go1.16.6.linux-amd64.tar.gz to /usr/local:

```
tar -C /usr/local -xzf go1.16.6.linux-amd64.tar.gz
```

Add /usr/local/go/bin to the PATH environment variable. You can do this by adding the following line to your \$HOME/.profile or /etc/profile (for a system-wide installation):

```
export PATH=$PATH:/usr/local/go/bin
```

Verify that you've installed Go by running the command below:

```
go version
```



## Build the AvalancheGo Image

Clone the AvalancheGo repository:

```
git clone https://github.com/ava-labs/avalanchego.git
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchego.git
```

You can find more about SSH and how to use it [here](#). :::

Build the image into docker:

```
cd avalanchego
./scripts/build_image.sh
```

To check the build image run the command below:

```
docker images
```



The image should be tagged as `avaplatform/avalanchego:COMMIT`, where `COMMIT` is the shortened commit of the Avalanche source it was built from. In our case it is `254b53da`.

## Setting Up and Running Chainlink Node

## Dependencies

1. Docker CE
2. Smartcontract/chainlink v0.10.3
3. AvalancheGo >= 1.4.5
4. PostgreSQL

## Steps to Run Chainlink Node

### 1. Run AvalancheGo

The first step is to run AvalancheGo within docker which will map the TCP ports 9650 and 9651 in the container to the same ports on the Docker host.

Use the command below to run the AvalancheGo image within Docker:

```
docker run --name avalanchego-chainlink -d -p 9650:9650 -p 9651:9651 -v /root/.avalanchego avaplatform/avalanchego:91599fea /avalanchego/build/avalanchego --network-id=fuji --http-host=
```

- --name assign a name to the container
- -d specifies detached mode
- -p specifies the port number
- -v specifies the docker host location to store the container volume data
- /avalanchego/build/avalanchego --network-id=fuji is the command to start the AvalancheGo under Fuji test network

Verify that the AvalancheGo node is started and running:

```
docker ps
```

This will list the AvalancheGo container status:



Also, you can check by requesting a CURL command and seeing if it returns JSON data, as it does here:

```
curl -X POST --data '{ "jsonrpc": "2.0", "id": 1, "method": "info.isBootstrapped", "params": { "chain": "C" } }' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```



### 2. Run PostgreSQL

The next step is to run PostgreSQL within a Docker container

Use the command below to run PostgreSQL. The POSTGRES\_PASSWORD and POSTGRES\_USER are the environment variables used to set up the PostgreSQL superuser and its password:

```
docker run --name pgchainlink -e POSTGRES_PASSWORD=chainlink -e POSTGRES_USER=chainlink -d -p 5432:5432 -v /root/postgres-data:/var/lib/postgresql/data postgres
```

- -name assign name to the container
- -e specifies the environment variables used for the container
- -p specifies the port number used for the container
- -v specifies the docker host location to store the container volume data

To verify that PostgreSQL is running, use the command below:

```
docker ps
```

This will list the Postgres container status:



## Run the Chainlink Node

The final step is to run the Chainlink node within the Docker container. Before we start running it, we need to do some basic setup which is required for Chainlink node.

1. Create a local directory to store Chainlink data
2. Create a .env file with the environment variables used to access the container.

3. Create a .api file with API email & password which is used to expose the API for GUI Interface
4. Create a .password file which holds the wallet password and is used to unlock the keystore file generated by you

Create a directory to store the Chainlink data:

```
mkdir ~/.chainlink-avalanche
```

Create a .env file to set the container's environment variables:

```
$ echo "ROOT=/chainlink
LOG_LEVEL=debug
ETH_CHAIN_ID=43113
MIN_OUTGOING_CONFIRMATIONS=2
LINK_CONTRACT_ADDRESS=0x0b9d5d9136855f6FEC3c0993feE6E9CE8a297846
CHAINLINK_TLS_PORT=0
SECURE_COOKIES=false
GAS_UPDATER_ENABLED=true
ALLOW_ORIGINS=*
ETH_URL=ws://$CHAINLINK_HOST:9650/ext/bc/C/ws
DATABASE_URL=postgresql://$USERNAME:$PASSWORD@$HOST:5432/chainlink?sslmode=disable" > ~/.chainlink-avalanche/.env
```

Create a .api file to expose credentials for the API and GUI interfaces:

```
$ echo "youremailaddress@yourcompany.com
$PASSWORD" > ~/.chainlink-avalanche/.api
```

Create a .password file for wallet password:

```
echo $PASSWORD > ~/.chainlink-avalanche/.password
```

#### NOTE

Don't forget to replace the \$CHAINLINK\_HOST, \$HOST, \$USERNAME and \$PASSWORD with actual values.

**Note:** The password needs to be at least 12 characters and must contain three lowercase, uppercase, numbers, and symbols.

Finally, run the Chainlink node.

Use the command below:

```
docker run -d --name chainlink-avalanche-node -p 6688:6688 -v ~/.chainlink-avalanche:/chainlink -it --env-file=/root/.chainlink-avalanche/.env smartcontract/chainlink:0.10.3 local n -p /chainlink/.password -a /chainlink/.api
```

To verify the Chainlink node is running:

```
docker ps
```

This will list the Chainlink node container status:



## Setup a Chainlink Job

Before we set up a job in the Chainlink node, we need to create an external adaptor which will gather the real-world data that interests us and provide this data to our dapp. For that, we have created a simple API based external adaptor written with NodeJS. [Click to download simple API external Adaptor](#). Please follow the README.md file to install and start the adaptor.

## Login to Chainlink GUI

You can now connect to your Chainlink node's UI interface by navigating to <http://localhost:6688>. If using a VPS, you can create a SSH tunnel to your node for 6688:localhost:6688 to enable connectivity to the GUI. Typically this is done like this:

```
ssh -i $KEY $USER@$REMOTE-IP -L 6688:localhost:6688 -N
```

Access <http://localhost:6688> in your favorite browser, and this will return to the Chainlink login page:



## Create a New Bridge

First, create a new bridge which will point to the external adaptor listening address, which is in our case [http://<\\$HOST>:8081](http://<$HOST>:8081)



## Create a New Job

The next step is to create a new job in the Chainlink node.

### Using Type: "web"

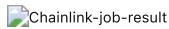
Below is a code snippet for the job specification to test the job within the Chainlink GUI. This can be done by using the "type": "web" . Please check the Chainlink official [documentation](#) for more details. Later, in this section we also cover the "type": "runlog" which will be used to integrate Chainlink with on-chain contracts.

```
{
 "name": "Avalanche NodeJS adapter test",
 "initiators": [
 {
 "type": "web"
 }
],
 "tasks": [
 {
 "type": "avalanchenodejs",
 "params": {
 "chain": "P",
 "method": "platform.getCurrentSupply",
 "params": {}
 }
 }
]
}
```

Now, in the jobs section, the newly created *Avalanche NodeJS adapter test* will be listed. Select that and click the Run button to test and see the result.



Then, click on the Runs tab, and select the job id to see the result. It should be in *Completed* status:



### Using Type: "runlog"

The [RunLog](#) initiator is the easiest initiator to use when integrating Chainlink with on-chain contracts.

```
{
 "name": "Avalanche NodeJS adapter",
 "initiators": [
 {
 "type": "runlog",
 "params": {
 "address": "0xac830beb7a2f1ced128e347e6b9a37dcc2e971b7"
 }
 }
],
 "tasks": [
 {
 "type": "avalanchenodejs",
 "params": {
 "chain": "P",
 "method": "platform.getCurrentSupply",
 "params": {}
 }
 },
 {
 "type": "jsonparse"
 },
 {
 "type": "ethuint256"
 },
 {
 "type": "ethtx"
 }
]
}
```

- `jsonparse`, `ethuint256` & `ethtx` are core adaptors and are executed synchronously. For more details, you will find documentation for each adapter's usage [here](#).
- `runlog` - By adding the address parameter, you make the event filter of the RunLog initiator more restrictive, only listening for events from that address, instead of any address. By adding the requesters parameter, you only allow requests to come from an address within the array. Please have a look at this [video](#) to learn how to get your oracle address.

## Smart Contracts

This section will explain the smart contracts we have written for communicating with the Chainlink node through oracle address and job id.

We have used the solidity and compiler version 0.4.24 for this tutorial. For more details please refer to [link](#) on how to compile the solidity program.

To deploy the smart contracts go to the Deploy & Run Transactions Tab.

There are 3 type of environments Remix can be plugged to:

- JavaScript VM
- Injected Web3
- Web3 Provider

Both Injected Web3 and Web3 Provider require the use of an external tool.

An external tool for Injected provider is MetaMask. Some external tools used with Web3 provider are a Truffle Ganache-CLI, Hardhat node, or an Ethereum node itself.

In this tutorial, we have used the Injected Web3 as environment and loaded the MetaMask with Avalanche Fuji Testnet. Please have a look at this [section](#) you can see our Avalanche Fuji Testnet Address has been chosen in the Account dropdown. For more details please refer to [link](#) on how to run & deploy the smart contracts.

Below is the code snippet which you can copy and paste into the Remix IDE and then compile and deploy as suggested above.

```
pragma solidity 0.4.24;

import "https://github.com/smartcontractkit/chainlink/contracts/src/v0.4/ChainlinkClient.sol";
import "https://github.com/smartcontractkit/chainlink/contracts/src/v0.4/vendor/Ownable.sol";

contract ATestnetConsumer is ChainlinkClient, Ownable {
 uint256 constant private ORACLE_PAYMENT = 1 * LINK;
 uint256 public supply;

 event RequestAvaxSupplyFulfilled(
 bytes32 indexed requestId,
 uint256 indexed supply
);

 constructor() public Ownable() {
 setChainlinkToken(0x0b9d5d9136855f6FEC3c0993feE6E9CE8a297846);
 }

 function requestCurrentAvaxSupply(address _oracle, string _jobId)
 public
 onlyOwner
 {
 Chainlink.Request memory req = buildChainlinkRequest(stringToBytes32(_jobId), this, this.fulfillAvaxSupply.selector);
 req.add("chain", "P");
 req.add("method", "platform.getCurrentSupply");
 req.add("path", "supply");
 sendChainlinkRequestTo(_oracle, req, ORACLE_PAYMENT);
 }

 function fulfillAvaxSupply(bytes32 _requestId, uint256 _supply)
 public
 recordChainlinkFulfillment(_requestId)
 {
 emit RequestAvaxSupplyFulfilled(_requestId, _supply);
 supply = _supply;
 }

 function getChainlinkToken() public view returns (address) {
 return chainlinkTokenAddress();
 }

 function stringToBytes32(string memory source) private pure returns (bytes32 result) {
 bytes memory tempEmptyStringTest = bytes(source);
 if (tempEmptyStringTest.length == 0) {
 return 0x0;
 }

 assembly { // solhint-disable-line no-inline-assembly
 result := mload(add(source, 32))
 }
 }
}
```

```
 }
}

}
```

## Deployed Smart Contracts



Add tokens to the Avalanche Fuji Testnet so that you can transfer some LINKs to the deployed blockchain address to perform the transactions. Please refer to [link](#) on how to add custom tokens.

Use the [Avalanche Faucet](#) and send some LINK tokens to the Fuji Testnet wallet address.

Then transfer LINKS to the deployed blockchain address to perform the transaction.



Now, call the `requestCurrentAvaxSupply` method on the deployed blockchain with params `oracle_address` & job id.

:::info The oracle address should be *your own oracle address*. Please have a look at this [video](#) to learn how to get your own oracle address. :::

- `oracle_address - 0xaC830Beb7a2f1cED128e347e6B9A37DCc2e971B7`
- `job id - 52c3344f35eb4f2e93343810199ab313"`



Now, you can check the Runs tab in Chainlink GUI it will show that the job has triggered and completed:



Then, click the supply button on the deployed contract it will return the current AVAX supply value:



## Conclusion

Having completed this tutorial, you are now no longer restricted to using only the oracles that Chainlink provides; now you can make your own!

You know how to setup a Chainlink node with the Avalanche Fuji Testnet and create dapp smart contracts to connect to the Chainlink node. You can follow this methodology to gather any real-world information that your dapp needs just by following this tutorial and designing your Chainlink adapter to gather different information.

As a final word, remember that in this tutorial, we describe each step of setting up the environment by hand, but in order to make your life easier now that you know how it works, we have included a set of Ansible scripts to complete this process automatically.

## Resources

Here is a list of resources that can give you a detailed idea of what is mentioned in this tutorial.

1. [Ansible installation and basics](#)
2. [Docker installation and basics](#)
3. [Learn how to write smart contracts with solidity](#)
4. [Learn NodeJS and installation](#)
5. [Learn Modern JavaScript](#)
6. [Learn about Chainlink](#)
7. [This is a useful documentation from Chainlink to setup and run a local Chainlink node](#)
8. [Here is a good YouTube video by Chainlink on how to running a local Chainlink node](#)
9. [Avalanche Testnet LINK Token details](#)
10. [AvalancheGo setup details](#)
11. [How to add custom tokens in MetaMask](#)

## Reddev Infrastructure Automation Repository

Setup requirements:

1. Ansible v2.9.9 or later
2. Get Vultr.com API key from <https://my.vultr.com/settings/#settingsapi>
3. Create file at `~/.vultr.ini` and fill in your API key like so:

```
[reddev]
key = quertyuiop123456789
timeout = 60
```

## Development Server Creation with VPS

1. Run `ansible-playbook -i inventory tasks/createvultrvps.yml`
2. Follow directions to create new development server in Vultr
3. Run `ansible-playbook -i inventory tasks/provision.yml`
4. Follow directions to provision the newly created development server

## Development Server Creation Within Same Host

1. Run `ansible-playbook tasks/provision_localhost.yml`

## Chainlink External Adaptor

Adaptor for communicating smart contract with real world data

```
cd chainlink-external-adaptor/
npm install
```

### Start / Stop Server

```
npm run startServer - To start the server npm run stopServer - To stop the server
```

Usage

```
curl -X POST -H "content-type:application/json" "http://localhost:8081/" --data '{ "id": 1, "data": {"chain": "P", "method": "platform.getCurrentSupply","params": {} }}'
```

## Tutorial: Avalanche Signature Verification in a dapp

### Introduction

This tutorial will show you how to use an Avalanche C-Chain dapp to verify the signature of a message like this that has been signed using the Avalanche Wallet.



We at [red-dev](#) needed to do this for our current software project under development, [RediYeti](#). We have a use-case where we need to verify ownership of an Avalanche X-Chain address before the dapp sends funds related to this address. To prevent fraud, the verification must take place inside of the dapp.

If you need to implement the same kind of signature verification in your dapp, you will find this tutorial useful. You may also find this tutorial useful if you would like to learn how Avalanche signatures work and how cryptography is implemented on the X-Chain and on the C-Chain.

We have included code snippets throughout, and you can find the entire project [here](#). Many steps are involved, but once you understand how they fit together, you will have a deeper understanding of how this aspect of Avalanche—and indeed cryptocurrencies in general—works.

### Audience

To get the most out of this tutorial, you will need to have a basic understanding of JavaScript, Node, Solidity, and how to develop dapps in Avalanche. You should also know the basics of public key cryptography. Avalanche uses Elliptic Curve Cryptography (ECC) as do Bitcoin, Ethereum, and many others. The ECC algorithm used for digital signatures is called ECDSA (Elliptic Curve Digital Signature Algorithm). If you do not yet know about these topics, see the **Resources** section at the end for links to learn more.

### Why Are Digital Signatures Important?

A digital signature system allows you to generate your own private/public key pair. You can then use the private key to generate digital signatures which let you prove that (1) you are the owner of the public key and (2) that the signed message has not been altered—both without having to reveal the private key. With cryptocurrencies, keeping your private key secret is what lets you hold onto your funds, and signing messages is how you transfer funds to others, so digital signatures are indeed foundational to Avalanche.

### Overview

At the very highest level, here is an overview of the process we will take you through in this tutorial. First we use a simple webapp to gather the three inputs: signing address, message, and signature. We extract the cryptographic data from them that will be passed into the dapp to verify the signature.

The dapp then verifies in two steps. First, it makes sure that all the values passed to it are provably related to each other. Then, assuming that they are, it uses the [elliptic-curve-solidity](#) library, which we have slightly modified to work with Avalanche, to verify the signature.

The dapp returns its result to the webapp which displays it. Of course, in your dapp, you will want to take some more actions in the dapp based on the result, depending on your needs.

(**Note:** If you're already a Solidity coder, you might think that there is an easier way to do this using the EVM's built-in function `ecrecover`. However, there is one small hitch that makes using `ecrecover` impossible: it uses a different hashing method. While Avalanche uses SHA-256 followed by ripemd160, the EVM uses Keccak-256.)

We have set up a [demo webpage here](#).

## Requirements

MetaMask needs to be installed on your browser, and you need to be connected to the Avalanche Fuji test network (for this tutorial). You can add a few lines of codes to check if your browser has MetaMask installed, and if installed, then to which network you are connected. For instance:

```
function checkMetamaskStatus() {
 if((window as any).ethereum) {
 if((window as any).ethereum.chainId != '0xa869') {
 result = "Failed: Not connected to Avalanche Fuji Testnet via MetaMask."
 }
 else {
 //call required function
 }
 }
 else {
 result = "Failed: MetaMask is not installed."
 }
}
```

## Dependencies

1. NodeJS v8.9.4 or later.
2. AvalancheJS library, which you can install with `npm install avalanche`
3. Elliptic Curve library, which can be installed with `npm install elliptic`
4. Ethers.js library, which can be installed with `npm install ethers`

## Steps that Need to Be Performed in the Webapp

To verify the signature and retrieve the signer X-Chain address, you first need to extract cryptographic data from the message and signature in your webapp, which will then be passed to the dapp. (The example code uses a **Vue** webapp, but you could use any framework you like or just Vanilla JavaScript.) These are:

1. The hashed message
2. The r, s, and v parameters of the signature
3. The x and y coordinates of the public key and the 33-byte compressed public key

We extract them using JavaScript instead of in the dapp because the Solidity EC library needs them to be separated, and it is easier to do it in JavaScript. There is no security risk in doing it here off-chain as we can verify in the dapp that they are indeed related to each other, returning a signature failure if they are not.

### 1. Hash the Message

First, you will need to hash the original message. Here is the standard way of hashing the message based on the Bitcoin Script format and Ethereum format:

**sha256(length(prefix) + prefix + length(message) + message)**

The prefix is a so-called "magic prefix" string `0x1AAvalanche Signed Message:\n`, where `0x1A` is the length of the prefix text and `length(message)` is an integer of the message size. After concatenating these together, hash the result with `sha256`. For example:

```
function hashMessage(message: string) {
 let mBuf: Buffer = Buffer.from(message, 'utf8')
 let msgSize: Buffer = Buffer.alloc(4)
 msgSize.writeUInt32BE(mBuf.length, 0)
 let msgBuf: Buffer = Buffer.from(`0x1AAvalanche Signed Message:\n${msgSize}${message}`, 'utf8')
 let hash: Buffer = createHash('sha256').update(msgBuf).digest()
 let hashhex: string = hash.toString('hex')
 let hashBuff: Buffer = Buffer.from(hashhex, 'hex')
 let messageHash: string = '0x' + hashhex
 return {hashBuff, messageHash}
}
```

### 2. Split the Signature

Avalanche Wallet displays the signature in CB58 Encoded form, so first you will need to decode the signature from CB58.

Then, with the decoded signature, you can recover the public key by parsing out the r, s, and v parameters from it. The signature is stored as a 65-byte buffer `[R || S || V]` where `V` is 0 or 1 to allow public key recoverability.

Note, while decoding the signature, if the signature has been altered, the `cb58Decode` function may throw an error, so remember to catch the error. Also, don't forget to import `bintools` from AvalancheJS library first.

```
function splitSig(signature: string) {
 try{
 let bintools: Bintools = BinTools.getInstance()
 let decodedSig: Buffer = bintools.cb58Decode(signature)
 const r: BN = new BN(bintools.copyFrom(decodedSig, 0, 32))
 const s: BN = new BN(bintools.copyFrom(decodedSig, 32, 64))
 const v: number = bintools.copyFrom(decodedSig, 64, 65).readUIntBE(0, 1)
 const sigParam: any = {
 r: r,
```

```

 s: s,
 v: v
 }
 let rhex: string = '0x' + r.toString('hex') //converts r to hex
 let shex: string = '0x' + s.toString('hex') //converts s to hex
 let sigHex: Array<string> = [rhex, shex]
 return {sigParam, sigHex}
}
catch{
 result = "Failed: Invalid signature."
}
},

```

### 3. Recover the Public Key

The public key can be recovered from the hashed message, r, s, and v parameters of the signature together with the help of Elliptic Curve JS library. You need to extract x and y coordinates of the public key to verify the signature as well as the 33-byte compressed public key to later recover the signer's X-Chain address.

```

function recover(msgHash: Buffer, sig: any) {
 let ec = new EC('secp256k1')
 const pubk: any = ec.recoverPubKey(msgHash, sig, sig.v)
 const pubkx: string = '0x' + pubk.x.toString('hex')
 const pubky: string = '0x' + pubk.y.toString('hex')
 let pubkCord: Array<string> = [pubkx, pubky]
 let pubkBuff: Buffer = Buffer.from(pubk.encodeCompressed())
 return {pubkCord, pubkBuff}
}

```

Here is the full code for verification, including the call to the dapp function `recoverAddress` at the end, which we will cover next:

```

async function verify() {
 //Create the provider and contract object to access the dapp functions
 const provider: any = new ethers.providers.Web3Provider((window as any).ethereum)
 const elliptic: any = new ethers.Contract(contractAddress.Contract, ECArtifact.abi, provider)
 //Extract all the data needed for signature verification
 let message: any = hashMessage(msg)
 let sign: any = splitSig(sig)
 let publicKey: any = recover(message.hashBuff, sign.sigParam)
 //prefix and hrp for Bech32 encoding
 let prefix: string = "fuji"
 let hrp: Array<any> = []
 for (var i=0; i<prefix.length; i++) {
 hrp[i] = prefix.charCodeAt(i)
 }
 //Call recoverAddress function from dapp. xchain and msg are user inputs in webapp
 const tx: string = await elliptic.recoverAddress(message.messageHash, sign.sigHex, publicKey.pubkCord, publicKey.pubkBuff,
msg, xchain, prefix, hrp)
 result = tx
}

```

### Recover the Signer X-Chain Address in Dapp

In the dapp, receive as a parameter the 33-byte compressed public key to recover the X-Chain Address.

Addresses on the X-Chain use the Bech32 standard with an Avalanche-specific prefix of **X-**. Then there are four parts to the Bech32 address scheme that follow.

1. A human-readable part (HRP). On Avalanche Mainnet this is `avax` and on Fuji testnet it is `fuji`.
2. The number 1, which separates the HRP from the address and error correction code.
3. A base-32 encoded string representing the 20 byte address.
4. A 6-character base-32 encoded error correction code.

Like Bitcoin, the addressing scheme of the Avalanche X-Chain relies on the **secp256k1** elliptic curve. Avalanche follows a similar approach as Bitcoin and hashes the ECDSA public key, so the 33-byte compressed public key is hashed with **sha256** first and then the result is hashed with **ripemd160** to produce a 20-byte address.

Next, this 20-byte value is converted to a **Bech32** address.

The `recoverAddress` function is called in the dapp from the webapp.

`recoverAddress` takes the following parameters:

- `messageHash`—the hashed message
- `r`—r and s value of the signature
- `publicKey`—x and y coordinates of the public key
- `pubk`—33-byte compressed public key
- `xchain`—X-Chain address
- `prefix`—Prefix for Bech32 addressing scheme (AVAX or Fuji)
- `hrp`—Array of each unicode character in the prefix

Then it performs the following steps:

1. Gets the 20-byte address by hashing the 33-byte compressed public key with sha256 followed by ripemd160.
2. Calls the Bech32 functions to convert the 20-byte address to Bech32 address (which is the unique part of the X-Chain address).
3. Verifies that the extracted X-Chain address matches with the X-Chain address from the webapp.
4. If X-Chain Address matches then validates the signature.
5. Returns the result.

Here is the `recoverAddress` function that does all this:

```
function recoverAddress(bytes32 messageHash, uint[2] memory rs, uint[2] memory publicKey, bytes memory pubk, string memory xchain, string memory prefix, uint[] memory hrp) public view returns(string memory){
 bool signVerification = false;
 string memory result;
 bytes32 sha = sha256(abi.encodePacked(pubk));
 bytes20 ripesha = ripemd160(abi.encodePacked(sha));
 uint[] memory rp = new uint[](20);
 for(uint i=0;i<20;i++) {
 rp[i] = uint(uint8(ripensha[i]));
 }
 bytes memory pre = bytes(prefix);
 string memory xc = encode(pre, hrp, convert(rp, 8, 5));
 if(keccak256(abi.encodePacked(xc)) == keccak256(abi.encodePacked(xchain))) {
 signVerification = validateSignature(messageHash, rs, publicKey);
 if(signVerification == true) {
 result = "Signature verified!";
 }
 else {
 result = "Signature verification failed!";
 }
 }
 else {
 result = string(abi.encodePacked("Failed: Addresses do not match. Address for this signature/message combination should
be ", xc));
 }
 return result;
}
```

Let's take a closer look at its supporting functions and key features.

### Bech32 Encoding

We have ported Bech32 to Solidity from the [Bech32 JavaScript library](#). There are four functions, `polymod`, `prefixChk`, `encode` and `convert`, used to convert to Bech32 address.

```
bytes constant CHARSET = 'qpzry9x8gf2tvdw0s3jn54khce6mua7l';

function polymod(uint256 pre) internal view returns(uint) {
 uint256 chk = pre >> 25;
 chk = ((pre & 0xffffffff) << 5)^(-((chk >> 0) & 1) & 0x3b6a57b2) ^
 (-((chk >> 1) & 1) & 0x26508e6d) ^
 (-((chk >> 2) & 1) & 0xleal119fa) ^
 (-((chk >> 3) & 1) & 0x3d4233dd) ^
 (-((chk >> 4) & 1) & 0x2a1462b3);
 return chk;
}

function prefixCheck(uint[] memory hrp) public view returns (uint) {
 uint chk = 1;
 uint c;
 uint v;
 for (uint pm = 0; pm < hrp.length; ++pm) {
 c = hrp[pm];
 chk = polymod(chk) ^ (c >> 5);
 }
 chk = polymod(chk);
 for (uint pm = 0; pm < hrp.length; ++pm) {
 v = hrp[pm];
 chk = polymod(chk) ^ (v & 0x1f);
 }
 return chk;
}

function encode(bytes memory prefix, uint[] memory hrp, uint[] memory data) public view returns (string memory) {
 uint256 chk = prefixCheck(hrp);
 bytes memory add = '1';
 bytes memory result = abi.encodePacked(prefix, add);
 for (uint pm = 0; pm < data.length; ++pm) {
 result = abi.encodePacked(result, data[pm]);
 }
 return result;
}
```

```

 uint256 x = data[pm];
 chk = polymod(chk) ^ x;
 result = abi.encodePacked(result, CHARSET[x]);
 }
 for (uint i = 0; i < 6; ++i) {
 chk = polymod(chk);
 }
 chk ^= 1;
 for (uint i = 0; i < 6; ++i) {
 uint256 v = (chk >> ((5 - i) * 5)) & 0x1f;
 result = abi.encodePacked(result, CHARSET[v]);
 }
 bytes memory chainid = 'X-';
 string memory s = string(abi.encodePacked(chainid, result));
 return s;
}

function convert(uint[] memory data, uint inBits, uint outBits) public view returns (uint[] memory) {
 uint value = 0;
 uint bits = 0;
 uint maxV = (1 << outBits) - 1;
 uint[] memory ret = new uint[](32);
 uint j = 0;
 for (uint i = 0; i < data.length; ++i) {
 value = (value << inBits) | data[i];
 bits += inBits;
 while (bits >= outBits) {
 bits -= outBits;
 ret[j] = (value >> bits) & maxV;
 j += 1;
 }
 }
 return ret;
}

```

### Verify X-Chain Address

It is a simple step, but it is very important to check to see if the extracted X-Chain address from the public key matches with the X-Chain address that was passed from the webapp. Otherwise, you may have a perfectly valid message signature but for a *different* X-Chain address than the webapp requested. Only if they match can you proceed to verify the signature. Otherwise, return an error message.

### Validate Signature

For verifying the signature, we start with this [Solidity project on Elliptic Curve](#).

However, this project uses the **secp256r1** curve. As Avalanche uses **secp256k1** curve, we need to modify the constant values based on this curve. (**Note:** Since modifying cryptographic functions is risky, these are the only modifications we have made.) The constants now look like this:

```

uint constant a = 0;
uint constant b = 7;
uint constant gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798;
uint constant gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8;
uint constant p = 0xFFEEFFFC2F;
uint constant n = 0xFFFFFFFFFFFFFFFFFFFFFFFEBAEDCE6AF48A03BBFD25E8CD0364141;

```

The key function we need is **validateSignature**.

```
function validateSignature(bytes32 messageHash, uint[2] memory rs, uint[2] memory publicKey) public view returns (bool)
```

**validateSignature** takes the same first three parameters as `recoverAddress`:

- `messageHash`—the hashed message
- `rs`—`r` and `s` value of the signature
- `publicKey`—`x` and `y` coordinates of the public key

### Finishing Up

After performing these tests, the dapp returns its decision whether the signature is valid or not to the webapp, and the webapp is then responsible for showing the final output to the user. As we mentioned above, in your dapp, you will probably want to take further actions accordingly.

### Resources

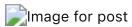
Here are some resources that can use to teach yourself the subjects you need in order to understand this tutorial.

1. This is a useful documentation from Ava Labs on cryptographic primitives: <https://docs.avax.network/build/references/cryptographic-primitives>
2. Here is a great YouTube video by Connor Daly of Ava Labs on how to use Hardhat to deploy and run your smart contract on Avalanche network: <https://www.youtube.com/watch?v=UdzHxdfMKkE&t=1812s>

3. If you want to learn more on how the private/public keys and the wallets work, you may enjoy going through this awesome tutorial by Greg Walker:  
<https://learnmeabitcoin.com/technical/>
4. Andrea Corbellini has done great work explaining Elliptic Curve Cryptography in detail in her blog post: <https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/>

## 2022 Winners

On June 6, 2022, Avalanche called for developers to rise to an occasion: [a contest](#) for the best tutorials explaining to other devs how to launch Avalanche Subnets. Subnets are, of course, Avalanche's means of [scaling infinitely](#). Anyone was able to submit a written tutorial, with selected tutorials eligible to win some of the cool \$32,000 prize pool.



The judges selected the winners.

:::warning

These tutorials are published as a snapshot when they were written. For up-to-date information, please reach out to the owners of these projects.

:::

### Customizing Subnet-EVM Genesis File

- [Avalanche - Creating a Genesis File for Your Subnet](#)

### Environment for Local Subnet Development

- [Set up Your Development Environment for Local Subnet Development](#)

### Use an ERC-20 C-Chain Token as the Gas Fee Token

- [How to Use a ERC-20 C-Chain Token as the Gas Fee Token](#)

### Free Topic

- [Indexing Local Subnets with TheGraph](#)

A giant thank you to the winners, and another to all who participated.

## Indexing an Avalanche Local Subnet with The Graph

### Introduction

[Avalanche](#) is an open-source platform for launching decentralized applications and enterprise blockchain deployments in one interoperable, highly scalable ecosystem. Avalanche is the first decentralized smart contracts platform built for the scale of global finance, with near-instant transaction finality. Avalanche is a blockchain that promises to combine scaling capabilities and quick confirmation times through its Avalanche Consensus Protocol. It can process 4,500 TPS (transactions per second). For Ethereum, that number is 14 TPS.

Blockchains have traditionally been referred to as being slow and unscalable. Avalanche embraces an innovative approach to consensus that solve these problems without compromising on security.

Avalanche is a high-performance, scalable, customizable, and secure blockchain platform. It targets three 15 broad use cases:

- Building application-specific blockchains, spanning permissioned (private) and permissionless (public) deployments.
- Building and launching highly scalable and decentralized applications (dapps).
- Building arbitrarily complex digital assets with custom rules, covenants, and riders (smart assets).

### Avalanche Features 3 Built-in Blockchains

- Exchange Chain (X-Chain)
- Platform Chain (P-Chain)
- Contract Chain (C-Chain)

The P-chain is for platform management. It handles requests related to the validator, the Subnet, and the blockchain. The C-chain is for contract management. It is based on EVM; hence its API is almost identical to other EVM protocols. It has both RPC and WebSocket endpoints, and it handles requests for smart contracts. The X-chain is for asset exchange. It is Avalanche's native platform; it is used for creating and trading assets like AVAX and NFTs.

These 3 blockchains are secured by the Avalanche Primary Network with a special kind of Subnet.

The Avalanche Architecture is composed of:

- Subnetworks
- Virtual Machines

### The Graph Protocol

[The Graph](#) is an open-sourced indexing protocol for organising blockchain data and making it easily accessible using [GraphQL](#). This software collects, processes and stores data from various blockchain applications to facilitate efficient information retrieval. The Graph stored data into various indices called Subgraphs, allowing applications to query it. These queries are initiated using GraphQL, a language originally created by facebook. The Graph has the ability to query networks like Ethereum and IPFS. Anyone can build and publish open subgraphs.



## Prerequisites

### 1. NodeJS and Yarn

First, install the LTS (long-term support) version of [NodeJS](#). This is `18.x` at the time of writing. NodeJS bundles `npm`.

Next, install the [yarn](#) package manager:

```
npm install -g yarn
```

### 2. Git

To check the current Git version use:

```
git --version
```

This tutorial is created to serve as a guide to help developers setup an Avalanche Subnet and Index them using GraphQL. We are going to learn how to run a local network using the [Avalanche-cli](#) and deploy a basic smart contract using [Remix](#). Then Lastly we will be indexing our Subnet using [The Graph](#). This guide is an extension of the [Official Avalanche Documentation](#).

Please note that all command line inputs and sample codes are MacOs and Linux Based. Commands may vary for other operating systems.

In summary, we will be discussing the following:

1. Running an EVM Subnet on the Local Network using the Avalanche-cli
2. Deploying smart contracts with Remix
3. Indexing our Subnet using The Graph

## Running an EVM Subnet on the Local Network Using the Avalanche-Cli

We will be creating an EVM on our local machine to give us a basic feel on how a Subnet functions. The [Avalanche-CLI](#) is a novel tool that allow us to have a local network up in minutes.

### Installation

Open up you MacOs command line utility and run the following command

On your home directory, create a new directory and `cd <newdir>` into the directory. This is where we will be installing all our project dependencies.

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s
```

This command download the latest binary of the [Avalanche-CLI](#) to the current directory where it was executed.

`cd` into the `bin` folder and export the path variable

```
cd bin
export PATH=$PWD:PATH
```

This makes the `avalanche` command available globally. For more information about [environment-variables](#) and [avalanche-cli-commands](#) visit the respective links.

### Initialising a Default Subnet

We will be using the `avalanche subnet create` command line wizard to get our network running. ASAP. In the same directory where the binary was installed, run the following command

```
avalanche subnet create <SubnetName>
```

Substitute `<SubnetName>` with any preferred name of your choice but without spaces. For this tutorial we are going to call our Subnet `<fibrinNet>`.

```
avalanche subnet create fibrinNet
```

Since this command does not provide any arguments, you would need to walk through the configuration wizard to generate a `genesis file` for your network.

- Choose a Virtual Machine (VM): We are going to be making use of the `SubnetEVM`
- Pick a chain ID Every EVM based blockchain has a special parameter called a `chainID`. ChainIDs are expected to be unique values, you can check [chainlist.org](#) to see if your proposed chain ID is already in use. We will be making use of the chain ID `1970` (A pun on JavaScript dates...lol).
- Select a symbol for the native Subnet token

- Set fees: Select the low disk use / low throughput option
- Airdrop: default to airdrop 1 million tokens to provided address
- Add a custom precompile to modify the EVM: For this section, we will not be using a pre-compile script

The wizard won't customize every aspect of the Subnet-EVM genesis for you, we will be doing this in the subsequent sections.



To view the list of all created Subnets, just execute the following command

```
avalanche subnet list
```



### Deploying the Subnet Locally

To deploy the newly created Subnet locally, run the following command

```
avalanche subnet deploy <SubnetName>
```



When a Subnet is run locally, it starts a multi-node (5 node) Avalanche Network in the background.



### Configuring MetaMask

To test the functionality of the just created Subnet, go ahead and add the configuration details to [MetaMask](#). You can create a new MetaMask account by importing the private key `0x56289e99c94b6912bfcc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027` and start experiencing with this account.

I have a [GitHub Tutorial](#) that explains how to setup your local development environment including [MetaMask](#).

Lastly don't forget to stop the running local network

```
avalanche network stop <snapshotName>
```



### Deploying Smart Contracts with Remix

#### 1. Setting up Remix

We are going to be making use of an online IDE in compiling and deploying our smart contract code, a tool called [Remix](#).

Navigate to the [Remix](#) platform and import the test contract we will be making use of.



```
https://github.com/FibrinLab/example-subgraph/blob/master/contracts/Gravity.sol
```

```
:::info
```

This repo contains the official sample subgraph for the [gravatar](#) registry.

```
:::
```

#### 1. Deployment Steps

Compile and deploy the `Gravity` smart contract using the Local Subnet you just created. To do this select the `deploy` tab and choose `injected web3` from the dropdown. Please note that Remix automatically detects the appropriate compiler version and makes use of it to compile your contract.



Always make sure to confirm the Environment Chain ID is the same as that of your selected MetaMask account.

With this all set go ahead and deploy your smart contract on your local Subnet by clicking the `deploy` button. Approve the MetaMask request and pay the necessary gas fees.

If the deployment is successful, you should see something like this ==>



Please take note of the deployment address as we will be making use of it subsequently.



## Indexing Our Subnet Using the Graph

### 1. Installing Dependencies

The most efficient way to make use of The Graph in indexing our Subnet is to host a local Graph Node. This is pretty straightforward to setup once you got requirements up and running. This tutorial is an extension of the [Graph-Node](#) GitHub repository.

The following components are needed:

- o Interplanetary File System (IPFS) for hosting our files. [Installation](#) instructions.
- o PostgreSQL, a database management tool for keeping out data. [Installation](#) instructions.
- o Rust, we will be building and compiling The Graph Node using the cargo package manager. [Installation](#) instructions.

If the above installation instructions are followed correctly, you should have these tools up and running.



### 2. Deploying Resources

After successfully installing IPFS, initialise the daemon by running

```
ipfs init
```



Next run,

```
ipfs daemon
```



Its now time to get our database all set-up.

Run the following commands:

```
initdb -D .postgres
```

```
pg_ctl -D .postgres -l logfile start
```

```
createdb graph-node
```

### 3. Configuring up the Graph Node

Clone and build The Graph node folder

```
git clone https://github.com/graphprotocol/graph-node
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:graphprotocol/graph-node.git
```

You can find more about SSH and how to use it [here](#). :::

Build the folder by running

```
cargo build
```

If you have successfully installed Rust but `the command is not found`, you would need to setup some environmental variables. Running this command might help.

```
source $HOME/.cargo/env
```

Once all dependencies are up and running, run the following command to kick start the node.

```
cargo run -p graph-node --release -- \
--postgres-url postgresql://postgres:*fill-in-postgresql-username: *fill-in-postgresql-password @localhost:5432/graph-node \
--ethereum-rpc fuji:http://127.0.0.1:9650/ext/bc/*fill-in-your-blockchain-id/rpc \
--ipfs 127.0.0.1:5001
```

Before running the above make sure you replace the following: `*fill-in-postgresql-username ==> Your Database username.` `Defaults to [Postgres] `*fill-in-postgresql-password ==> Your Database password.`



If everything goes smoothly. You should get this.



#### 4. Deploying the subgraph

This is where things get interesting. Change directory into the `example-subgraph` folder

Clone the official subgraph repository and install all the dependencies

```
git clone https://github.com/graphprotocol/example-subgraph
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:graphprotocol/example-subgraph.git
```

You can find more about SSH and how to use it [here](#). :::

Next, generate the ABI typings

```
yarn
yarn codegen
```

Open you the `subgraph.yaml` file and make 2 (two) modifications under `datasources`.

- Switch the network to `local`



- Input the address of the deployed `Gravity` contract in the `address` field



Finally, run the following

```
yarn create-local
yarn deploy-local
```

Congratulations, you have successfully deployed a Sub-Graph on a locally deployed Subnet.



After successful deployment, your graph node would need a few minutes to scan all the nodes.



Once its done, open up the provided link in the browser.



Open up the link and try running a query by filling this into the query box.

```
query MyQuery {
 gravatars {
```

```
 id
 imageUrl
 displayName
 }
}
```

Watch the magic happen.



## Conclusion

In summary, we have deployed a Local Subnet using the Avalanche-cli. We further went ahead to deploy smart contracts, run a Graph node and Index our nodes using `The Graph`. How cool is that? lol.

Feel free to fork this repository and build great stuff.

Cheers and Happy Coding.

Akanimoh Osutuk

# Avalanche - Creating a Genesis File for Your Subnet

## Introduction

Hi there! In this tutorial, we will be learning how to create a custom genesis file for your Subnet.

We will start by deploying a very simple Subnet using [the Subnet wizard](#). Then we will learn about the genesis file and how to customize it for our needs.

Next, we will create a simple game!

We will create and deploy a new Subnet using our custom genesis file for the game. Finally, we will be deploying the Game contract to the Subnet.

The Game contract has a simple function called `play` that accepts `LUCK` tokens (native currency of our Subnet) and the contract will either lock the tokens or `mint` 2x the amount of tokens to the player. Yes, it's going to mint native tokens.

At the end of the tutorial, you will *hopefully* have a solid understanding of how to customize the genesis file of your Subnet.

I hope you enjoy this tutorial!

## Prerequisites

### MetaMask

You need to have [MetaMask](#) extension installed on your browser.

### Avalanche-CLI

Create a folder for Avalanche CLI.

```
mkdir -p ~/avalanche
```

Go to that directory and download the Avalanche CLI.

```
cd ~/avalanche
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s

> ava-labs/avalanche-cli info checking GitHub for latest tag
> ava-labs/avalanche-cli info found version: 0.1.3 for linux/amd64
> ava-labs/avalanche-cli info installed ./bin/avalanche
```

Go to the directory where you just downloaded the CLI and add it to your PATH. So you can run the CLI from anywhere.

```
cd ./bin/avalanche
export PATH=$PWD:$PATH
```

We're ready to deploy our first Subnet.

## Creating and Deploying a Simple Subnet

### What Is a Subnet?

A Subnet, or Subnetwork, is a dynamic subset of Avalanche Primary Network validators working together to achieve consensus on the state of one or more blockchains. Each blockchain is validated by exactly one Subnet. A Subnet can have and validate many blockchains. A validator may be a member of many Subnets.

Subnets are independent and don't share execution thread, storage or networking with other Subnets or the Primary Network, effectively allowing the network to scale up easily. They share the benefits provided by the Avalanche Protocol such as low cost and fast to finality.

A Subnet manages its own membership, and it may require that its constituent validators have certain properties. This is very useful, and we explore its ramifications in more depth below:

[Here is a great introduction to Subnets by Avalanche](#)

## Subnets

Subnets is a great technology! It's still advancing and a bit complicated. There is almost no tooling for Subnets and it's a bit difficult to work with them.

Avalanche has changed that! The Avalanche team has developed an amazing tool to make this process a breeze. Now you can deploy your own production-ready Subnet with just a few commands. The tool will guide you through configuring your Subnet and deploying it!

## Genesis File

The genesis file is a file that contains the initial configuration of the Subnet. Avalanche will generate a genesis file for you based on some parameters you provide. On the other hand, you can also create your own genesis file! This will allow you to have more control over the configuration of your Subnet.

## Creating a Simple Subnet

For this time, let's take the easy way out and use the CLI to deploy a Subnet to see how things work.

```
avalanche subnet create testSubnet
```

This command will run the wizard to create a Subnet.

There are six steps in the wizard:

### Choose Your VM

Choose SubnetEVM here

### Chain Id

Chain id is a unique identifier for the network. This value must be unique so check [chainlist](#) to see if the chain id is already in use.

### Token Symbol

Token symbol is the symbol of the native token used in the Subnet. For instance, on the C-Chain, the symbol is `AVAX` and on the Ethereum Mainnet, the symbol is `ETH`. You can use any symbol you want.

### Gas Configuration

This step will define the gas configuration of your network. You can choose a preset fee configuration or create your own. Keep in mind that more transaction throughput means more disk usage.

We will go with C-Chain default fees.

```
> Low disk use / Low Throughput 1.5 mil gas/s (C-Chain's setting)
```

### Airdropping Native Tokens

This step will define how you want to distribute the funds in your Subnet. You can choose the addresses and amounts you want to distribute. Let's go with the simplest case.

```
How would you like to distribute funds?
> Airdrop 1 million tokens to the default address (do not use in production)
```

### Adding a Custom Precompile to Modify the Evm

```
? Advanced: Would you like to add a custom precompile to modify the EVM?:
▶ No
Yes
Go back to previous step
```

Here you can set up a custom precompile to unlock some useful functions.

### Native Minting

This precompile allows admins to permit designated contracts to mint the native token on your Subnet. We will discuss this in more detail in the [contractNativeMinterConfig](#) section.

### CONFIGURE CONTRACT DEPLOYMENT WHITELIST

This precompile restricts who has the ability to deploy contracts on your Subnet. We will discuss this in more detail in the [contractDeployerAllowListConfig](#) section.

### CONFIGURE TRANSACTION ALLOW LIST

This precompile restricts who has the ability to make transactions on your Subnet. We will discuss this in more detail in the [txAllowListConfig](#) section.

However, we are not gonna do any of these yet. Choose `No` and proceed.

```
> Successfully created genesis
```

Congrats! You just created a Subnet! 🎉

## Deploying the Subnet

This is the most fun part. We will deploy the Subnet that we just created, and it's so easy to do, thanks to Avalanche CLI!

```
avalanche subnet deploy testSubnet

> ? Choose a network to deploy on:
▶ Local Network
 Fuji
 Mainnet
```

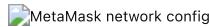
We will deploy the Subnet on a local network so choose `Local Network` here.

This process will install AvalancheGo (if not already installed) and start the network. This will take a couple of minutes so, this is a good time to refresh your coffee ☕

After this process is complete, you will see the MetaMask connection details in the terminal.

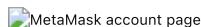
```
MetaMask connection details (any node URL from above works):
RPC URL: http://127.0.0.1:34249/ext/bc/8MiXurGFiuUfx3JzT4jXHVavgxDaBM4HSWotsGoeQMqRzcWWF/rpc
Funded address: 0x8db97C7EcE249c2b98bDC0226C4C2A57BF52FC with 1000000 (10^18) - private key:
56289e99c94b6912bfcc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027
Network name: testSubnet
Chain ID: 1234
Currency Symbol: TEST
```

You can use this configuration to add a new network to your MetaMask. Then you can import the account to which the airdrop funds were sent to into your MetaMask by importing the private key.



Notice that you will see the same private key if you've chosen `Airdrop 1 million tokens to the default address (do not use in production)` in the [airdrop step](#). Thus, the private key is exposed and anyone can access the account. For that reason, it's not a good idea to use this in production.

After importing the account on MetaMask, let's change the network to our Subnet.



How awesome is that? `TEST` is the native token of our Subnet and it's airdropped to our address. You can now deploy smart contracts and interact with the network.

## Understanding the Genesis File

Before starting, I'd like to acknowledge that this section is very similar to [AVAX docs - Customize a Subnet](#). However, I've made some changes to make it more comprehensive and easier to understand.

Subnet wizard is an awesome tool to customize your Subnet and might be a good starting point for you.

We have mentioned the [genesis file](#) above. The wizard generates this file for you but you can also create **your own genesis file!** This will allow you to customize the network more easily and in depth.

Let's take a look at the genesis file of our Subnet which the wizard created for us.

```
use your Subnet name instead of testSubnet
cat ~/.avalanche-cli/testSubnet_genesis.json
```

To access the genesis file, you can also use:

```
avalanche subnet describe testSubnet --genesis
```

Yes, the genesis file is in json format. The output should look like:

```
{
 "config": {
 "chainId": 1234,
 "homesteadBlock": 0,
 "eip150Block": 0,
 "eip150Hash": "0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",
 "eip155Block": 0,
 "eip158Block": 0,
 "byzantiumBlock": 0,
 "constantinopleBlock": 0,
 "petersburgBlock": 0,
 "istanbulBlock": 0,
```

Pretty scary, right? But it's not that bad as it seems. The genesis file is a json file that contains the configuration of the Subnet and the block header of the genesis block, which is the first block in the network.

Let's dive into the genesis file a little bit more.

## Config

**chainId**

The chain ID of the Subnet. We've set this to 1234 [when we created the Subnet](#)

## Hardforks

`eip150Block`, `eip150Hash`, `eip155Block`, `eip158Block`, `byzantiumBlock`, `constantinopleBlock`, `petersburgBlock`, `istanbulBlock`,  
`muirGlacierBlock`, `SubnetEVMTimestamp`

These are the blocks where the network has been adopted to the new protocol release. [Here is the full list](#).

When there is a new protocol release, to activate that protocol on your Subnet, you can add the configuration for the new protocol to the genesis file and point it to a block number in the future as the activation block number.

## Fee Config

gasLimit

The total amount of gas that can be used in a single block. Keep in mind that this impacts how much computation happens in one block. This is set to 8,000,000 in C-Chain. Also, the value represents the maximum amount of gas a single transaction can use.

**targetBlockRate**

The network aims to produce a new block in `targetBlockRate` seconds. This value is in **seconds**. If the network starts producing faster than this, [base fees are increased accordingly](#). Otherwise, if the network starts producing slower than this, [base fees are decreased accordingly](#). This value is set to `2` in C-Chain.

Page 2

The minimum base fee that can be used by a transaction. It also shows how much gas will cost to do native token transfers. The value is in wei and set to 25000000000 in C-Chain. It corresponds to 25 nAvax (gwei). You can use [Snowtrace unitconverter](#) to convert between units.

**targetGas**

The targeted amount of gas (including block gas cost) to consume within a rolling 10s window.

**baseFeeChangeDenominator**

The base fee can change over time. If the parent block used **more** gas than its target, the base fee should **increase**. Otherwise, if the parent block used **less** gas than its target, the base fee should **decrease**.

Here are the calculations:

If the parent block used **more** gas than its target:

```
baseFee = max(1, parentBaseFee * (parentGasUsed - parentGasTarget) / parentGasTarget / baseFeeChangeDenominator)
```

If the parent block used less gas than its target:

The `baseFeeChangeDenominator` is set to `36` in C-Chain. This value sets the base fee to increase or decrease by a factor of `1/36` of the parent block's base fee.

Setting this value to a larger value in

METHANE GAS COST

Each `Block` and `BlockGroup` bounds the root node of a block.

This value determines the block gas change rate depending on the `targetBlockRate`. If the parent block is produced at the `targetBlockRate`, the block gas cost will stay the same. If the parent block is produced at a **slower** rate, the block gas cost will **decrease**. If the parent block is produced at a **faster** rate, the block gas cost will **increase**. The amount of change is determined by the following formula:

(blockGasCostStep) \* (targetBlockRate - parent.block.produced\_time)

For example, if the `targetBlockRate` is set to 2 seconds, `blockGasCostStep` is set to 10 and the parent block was produced in 5 seconds. The block gas cost will decrease by 30:  $10 * (2 - 5) = -30$

This value is set to 200000 in C-Chain.

### **Validator Fee Recipient**

This configuration allows validators to specify a fee recipient

Use the following configuration to enable validators to receive fees. You can find more information from the Avalanche docs.

```
{
 "config": {
 "allowFeeRecipients": true
 }
}
```

## Precompiles

You can enable some precompiled contracts in the genesis file. These contracts provide God mode functionalities and are very useful for some cases. If you'd not like to use them, you can disable them by setting the config fields to null or by simply removing them from the genesis file.

```
contractDeployerAllowListConfig
```

This configuration allows you specify which addresses are authorized to deploy contracts. You can also customize this [in the last step of the Subnet wizard](#). If you'd like to restrict the contract deployer to a specific list of addresses, you can set the `contractDeployerAllowListConfig` to a JSON object with the following properties:

```
"contractDeployerAllowListConfig": {
 "blockTimestamp": 0,
 "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC","0x0000...", "... and more"]
}
```

Admin addresses can deploy new contracts and add new Admin and Deployer addresses. Precompiled contract is deployed to `0x020000000000000000000000000000000000000000000000000000000000000`. You can find more information from the [Avalanche docs](#).



We've done this part of the configuration [using the wizard](#). It's pretty straightforward. You can set the `alloc` to a JSON object with the following properties:

```
{
 "address_without_0x": {
 "balance": "balance"
 }
}
```

Two important things to notice here:

- The `address_without_0x` is the address of the account that will receive the tokens. Must **NOT** include the `0x` prefix.
  - Balance is the amount of tokens that will be allocated **in wei** to the account.

In our genesis file, the balance is set to `0xd3c21bcecceda1000000` which corresponds to `10000000000000000000000000000000` in decimal. You can also type in the balance in decimal. Notice that this number is in wei and it's equal to 1,000,000 AVAX. You can use [Snowtrace unitconverter](#) to convert between units.

## Creating a Custom Genesis File

We learned a lot! Now we can create our genesis file!

I'd like to create a Subnet where people can earn native tokens by playing a simple game.

The network doesn't need to be very fast. I'll set the transaction fees high because the game will mint native tokens to players so we should be burning some of them to help inflation.

To get started, create a new file named `genesis.json` and fill it with the following content:

This is a boilerplate genesis file. We are going to fill in the fields that we need.

## Setting the Chain Id

The first thing to do is to pick a unique [chain id](#) for our Subnet. We can check [chainlist](#) to find a chain id that is not used. I'll go with `321123` because it looks good.

```
{
 "config": {
 "chainId": 321123
 }
}
```

## Configuring Fees and Gas

I have an old computer so I want my Subnet to be - slow. So, I'll set the `gasLimit` to `5,000,000`, the `targetBlockRate` to 5 seconds and the `targetGas` to `10000000`.

Then I'll set the `minBaseFee` to 60 nAvax (qwei) which is 600000000000 wei to make sure that the transaction fees are high enough to help the network.

I'll set the `baseFeeChangeDenominator` to 50 because I don't want the base fee to change to be rapidly changing. Then I'll set the `minBlockGasCost` to 0, `maxBlockGasCost` to 5000000, and the `blockGasCostStep` to 100000 as I don't want the block gas cost to change too quickly.

The final configuration looks like:

```
{
 "config": {
 "feeConfig": {
 "gasLimit": 5000000,
 "targetBlockRate": 5,
 "minBaseFee": 6000000000,
 "targetGas": 1000000,
 "baseFeeChangeDenominator": 50,
 "minBlockGasCost": 0,
 "maxBlockGasCost": 5000000,
 "blockGasCostStep": 10000
 }
 }
}
```

Almost forgot! We should also update the `gasLimit` in the genesis block header.

```
{
 "config": {
 ...
 },
 "gasLimit": "5000000"
}
```

### Allocating Native Tokens

Let's allocate some native tokens for our address. You should use your own address here. `0x0000000b9af48743ef1188f3F20c9b8B90F52a5b` is my address.

```
{
 "alloc": {
 "0000000b9af48743ef1188f3F20c9b8B90F52a5b": {
 "balance": "0x3635C9ADC5DEA00000"
 }
 }
}
```

I'm being generous and allocating only 1000 tokens for myself. 1000 tokens corresponds to `10000000000000000000000000000000` in wei and that's `0x3635C9ADC5DEA00000` in hexadecimal. Keep in mind that you can also write this in decimal. I'm going to use the hexadecimal notation here because it looks better. You can use [Decimal to Hexadecimal converter](#) to convert between units.

### Minting Native Tokens

We will use [`contractNativeMinterConfig`](#) to mint native tokens in the game.

We will add our address as an Admin so we can add new Minter and Admin addresses in the future.

**Use your own address** instead of `0x0000000b9af48743ef1188f3F20c9b8B90F52a5b`.

```
{
 "config": {
 "contractNativeMinterConfig": {
 "blockTimestamp": 0,
 "adminAddresses": ["0x0000000b9af48743ef1188f3F20c9b8B90F52a5b"]
 }
 }
}
```

### Restricting Contract Deployers

We will deploy the contracts and players will use those contracts to play the game. So we don't need other people to deploy the contracts. We can restrict the smart contract deployers by using [`contractDeployerAllowListConfig`](#).

We will add our address as an Admin so we can add new Deployer and Admin addresses in the future.

**Use your own address** instead of `0x0000000b9af48743ef1188f3F20c9b8B90F52a5b`.

```
{
 "config": {
 "contractDeployerAllowListConfig": {
 "blockTimestamp": 0,
 "adminAddresses": ["0x0000000b9af48743ef1188f3F20c9b8B90F52a5b"]
 }
 }
}
```

Now we're done! The final config looks like:

## Creating and Deploying the Subnet Using the Custom Genesis File

We will use `genesis.json` to create a new Subnet. This part is pretty simple.

Let's first delete the old Subnet that we created using the wizard.

```
avalanche subnet delete testSubnet

> Deleted Subnet
```

Next, we will create a new Subnet using the genesis file

```
avalanche subnet create <SubnetName> --genesis <filepath>

avalanche subnet create game --genesis ./genesis.json
Use SubnetEVM

> Using specified genesis
> ✓ SubnetEVM
> Successfully created genesis
```

Now let's deploy it!

```
avalanche subnet deploy <SubnetName>
avalanche subnet deploy game
```

```

choose local network as deployment target

> MetaMask connection details (any node URL from above works):
> RPC URL: http://127.0.0.1:27200/ext/bc/HGwbjkQsvrqy2mncMWsA2YnrSWEybVmLcZJwz7nb5gNVvcPoi/rpc
> Funded address: 0x0000000b9af48743ef1188f3F20c9b8B90F52a5b with 1000
> Network name: game
> Chain ID: 321123
> Currency Symbol: TEST

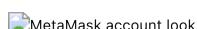
```

Awesomeness! Great work!

## Deploying the Game Contract

### MetaMask Configuration

Before going to the next step, make sure you've added the Subnet to MetaMask by following the same steps in the [Deploying the Subnet](#) section. This time you don't need to import the account as you're already using your own account. When adding the network, I've set the currency symbol to `LUCK`.



### MetaMask Gas Configuration ! IMPORTANT

Because we've set the base fee to 60 nAvax, you may need to increase the `Max priority fee` when confirming the transaction on MetaMask popup. Otherwise the transaction may fail.

Here is how:



If you're using the *Enhanced Gas Fee UI* which is an experimental feature in MetaMask, you can set the gas fee to *Aggressive* and that should work.



After making a few transactions, MetaMask should start automatically adjusting the gas fee to the optimal value.

### Deploying the Contract

We will use [remix](#) to deploy the game contract on the Subnet.

Go to the website and create a blank workspace.



Next, create a folder and under that folder, create two files with the following contents:



#### NativeMinterInterface.sol

This is the `ContractNativeMinter` interface. We know the contract is deployed at `0x0200000000000000000000000000000000000000000000000000000000000001`. [See more about it here.](#)

```

// (c) 2022-2023, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

interface NativeMinterInterface {
 // Set [addr] to have the admin role over the minter list
 function setAdmin(address addr) external;

 // Set [addr] to be enabled on the minter list
 function setEnabled(address addr) external;

 // Set [addr] to have no role over the minter list
 function setNone(address addr) external;

 // Read the status of [addr]
 function readAllowList(address addr) external view returns (uint256);

 // Mint [amount] number of native coins and send to [addr]
}

```

```
 function mintNativeCoin(address addr, uint256 amount) external;
 }
```

## Game.sol

A simple contract I made for the sake of this tutorial.

Basically, the player calls the `play` function with sending some `LUCK` tokens. The function generates a random number and by 50% chance, the player gets 2x the amount of tokens; otherwise, the tokens are stuck in the contract.

```
pragma solidity ^0.8.7;
import "./NativeMinterInterface.sol";

contract Game {

 // native minter is deployed at 0x020001
 NativeMinterInterface minter = NativeMinterInterface(0x0200);

 function random() internal view returns (uint) {
 // never use this random number generation in a real project
 // it's deterministic and easily hackable.
 return uint(keccak256(abi.encodePacked(block.timestamp)));
 }

 function play() external payable {
 uint256 randomNumber = random();

 if(randomNumber % 2 == 0){
 minter.mintNativeCoin(
 msg.sender,
 msg.value * 2
);
 }
 }
}
```

When you're done, it should look like this:



## Compiling and Deploying

After creating the files, we should compile `Game.sol`.

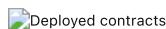


Go to the deploy tab, choose injected web3 as environment - connect your wallet

- then choose Game in the contract dropdown. Then click the deploy button.



Confirm the transaction with your MetaMask wallet.



Yay! We deployed a smart contract in our Subnet! That sounds fantastic!

## Setting Game Contract as a Minter

We're admin in `contractNativeMinterConfig`. Which means we can add new Minters to the network using `NativeMinterInterface`.

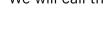
We already created `NativeMinterInterface.sol`

Go to remix and compile `NativeMinterInterface.sol` by following the same steps as `Game.sol`.



 Interface steps

After clicking the button, you should see the NATIVEMININTERFACE AT 0x02.. under the Deployed Contracts section.



Confirm the transaction with your MetaMask wallet. We can call this function because we've added our address as an admin in the `contractNativeMinterConfig` section.

Now the `Game.sol` contract is a minter and can mint native tokens!

## Play the Game

In the deploy tab:

- Set the value to 100 Ether or any other amount you want to play with.
- Find `Game` contract under deployed contracts section in remix.
- Call `play` function.
- Confirm the transaction with your MetaMask wallet.

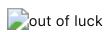


Then check your MetaMask wallet.



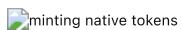
I first played with 10 and then 100 tokens. Seems like it's my lucky day! I already earned 110 LUCK tokens. As you can see, our smart contract **can mint native tokens!**

**Let's go all in!**



I'm out of luck!

As an admin in `NativeMinter` contract, I can mint native tokens as well! Let's mint ourselves 1000 LUCK tokens.



And my luck is back!



## Conclusion

This is the end of the tutorial. I hope you've enjoyed it as much as I've enjoyed writing it. Subnets are literally amazing and I'm very excited for the future of this technology.

You now have the ability to create your own Subnet and customize it to your needs. Go play with the parameters, try something new, and enjoy!

Please feel free to create an issue if you have any questions!

Thanks for reading :heart:

## Resources

- [AVAX Docs](#)
- [Explanation of genesis file](#)

# How to Use a ERC-20 C-chain Token as the Gas Fee Token

## Introduction

Purpose of this tutorial is to use a ERC-20 C-chain token as the gas fee token on a Subnet by deploying a bridge. We will not be using a bridge provider, instead we will be implementing our own bridge to truly understand how bridges work and how to modify them to our needs. Bridge we will be implementing is a [trusted](#) bridge and it uses [lock-mint](#) mechanism to transfer assets from C-chain to Subnet and [burn-release](#) mechanism from Subnet to C-chain.

:::caution

DISCLAIMER: The bridge implementation in this tutorial is a proof of concept and is not production ready.

:::

## Prerequisites

- Basic knowledge of [Precompiles](#).
  - We will be using [NativeMinter](#) precompile on our Subnet. Familiarity with precompiles and knowledge of NativeMinter precompile will be assumed.
- Having an up and running Subnet which uses NativeMinter precompile.
  - In this tutorial we will be using a local Subnet. Refer to [this](#), to deploy your local Subnet.
- Basic knowledge of [Hardhat](#).
  - We will be writing our code in a Hardhat development environment. We will write custom scripts to automate our job and add those scripts as tasks to hardhat.

- General knowledge of [EthersJS](#).
  - We will be interacting with both the Avalanche Fuji chain and our Subnet using EthersJS. We will be initializing providers, signers, contracts and interacting with contracts using EthersJS.
- General knowledge of [Solidity](#).
  - We will be writing our own Bridge and Token contracts using Solidity.

## General Concepts

Before writing any code, we should understand how bridges work and what type of bridges exist to better understand what we are actually doing. Reading Ethereum's official documentation about [bridges](#) is highly recommended.

### High Level Overview of Our Bridge Design

- A ERC-20 token on the C-chain and a native gas token on our Subnet that is mintable, using NativeMinter precompile.
- Bridge contracts on both chains with custom behaviors for ERC-20 and NativeMinter precompile.
- An off-chain relayer which is an application that listens to events on both chains. On these events, it sends transactions to other chain's bridge contract. For more detail refer to [Relayer](#) part.

### Trusted and Trustless Bridges

In short, bridges allow cross-chain transfers and there are 2 main types of bridges; Trusted and Trustless. Trusted bridges depend on an entity and have trust assumptions. Trustless bridges do not require trust assumptions to external entities but only to blockchain itself. To get more details refer to Ethereum's official documentation about [bridges](#).

### Lock-Mint and Burn-Release Mechanisms

Reason we have chosen these mechanisms is because our ERC-20 token might not be mintable and we are sure that our Subnet's gas token is mintable.

#### How Lock-Mint Works

- User deposits tokens to the bridge contract, effectively locking it.
- Relayer detects the transfer event and sends a transaction to the bridge contract that is on the other chain.
- Bridge contract on the other chain mints the token and sends it to the given address.

We are using the Lock-Mint mechanism when we bridge our token from C-chain to Subnet.

#### How Burn-Release Works

- User deposits tokens to the bridge contract, and the bridge contract sends them to the 0 address, effectively burning it.
- Relayer detects the transfer event and sends a transaction to the bridge contract that is on the other chain.
- Bridge contract on the other chain sends the token to the given address, effectively releasing it.

We are using the Burn-Release mechanism when we bridge our token from Subnet to C-chain.

:::caution

If bridge contract does not have enough tokens, it can not release them. Therefore, make sure that bridge is sufficiently funded to release intended amount. This was not a concern for lock-mint mechanism because bridge could always mint.

:::

## Building Blocks of Our Bridge

### Relayer

Relayer is an off-chain application that listens for events on both chains. Upon events it sends transactions to other chain's bridge contracts. It has the private key of the bridge contracts' admin account allowing it to mint or release tokens.

### Contracts

We will have bridge contracts on both chains. Users will send transactions to these contracts when they want to burn or lock their token on the respective chain. When a burn or lock happens these contracts emit an event for the relayer to observe. When the relayer observes the event on one chain, it will call bridge contract on the other chain with either mint or release function.

## Requirements

- [NodeJS](#) >= 16.0.0 must be installed.
- [npm](#) >= 8.15.0 must be installed.
- [Hardhat](#) >= 2.10.1 must be installed.

## Getting Started

### Initialize the Project

Let's start by initializing our workspace with [Hardhat](#).

To initialize project, run:

```
npx hardhat init
```

Select `Create a JavaScript project` and walk through creating the project.

### Create Contracts

Delete `Lock.sol` file

To use OpenZeppelin contracts in our contracts, run:

```
npm i @openzeppelin/contracts
```

### Create Token Contracts

Create a `Token` folder inside `contracts` folder

#### AVAX Token

Create `AvaxToken.sol` file inside `Token` folder

[AvaxToken.sol](#)

#### Native Minter Interface

Create `INativeMinter.sol` file inside `Token` folder

[INativeMinter.sol](#)

### Create Bridge Contracts

Create a `Bridge` folder inside `contracts` folder

#### AVAX Bridge

Create `AvaxBridge.sol` file inside `Bridge` folder

[AvaxBridge.sol](#)

#### Subnet Bridge

Create `SubnetBridge.sol` file inside `Bridge` folder

[SubnetBridge.sol](#)

### Compile Contracts

To make sure there are no problems with contracts, run:

```
npx hardhat compile
```

## Interact with Contracts

### Constants

First we will create a `constants` folder at the root of the project to store some general values.

Inside `constants` folder create `chains.js` and `nativeMinterAddress.js`.

[chains.js](#)

[nativeMinterAddress.js](#)

### Variables

Secondly we will create a `variables` folder at the root of the project to store some updated values such as contract addresses.

Inside the `variables` folder create `contractAddresses.js` but do not put anything in it. This file will be auto generated whenever we deploy some contracts.

### Utils

Then we will create a `utils` folder at the root of the project to define some general use functions.

Inside `utils` folder create `initProviders.js`, `initSigners.js` and `initContracts.js`

[initProviders.js](#)

[initSigners.js](#)

[initContracts.js](#)

### Scripts

We could directly create the off-chain relayer but it would be too hard to test it. Therefore, we are creating these helper scripts to quickly interact with contracts and test if relayer works as expected.

We are using [Hardhat Tasks](#) to run our scripts in this tutorial. We will be writing the script, updating the `hardhat.config.js` to add our new script as a hardhat task.

### Deploy Script

#### Write Deploy Script

Create `deploy.js` file inside `scripts` folder

### [deploy.js](#)

:::warning

Make sure that `BRIDGE_ADMIN` has an admin role for NativeMinter. So that it can allow `SubnetBridge` contract to mint native token.

:::

### **Update hardhat.config.js File**

To add `deploy` script as a hardhat task add following code inside `hardhat.config.js`

```
/* Import task from hardhat config */
const { task } = require("hardhat/config");
...
/* Import deploy function */
require("./scripts/deploy");
...
/* Create deploy task */
task(
 "deploy",
 "Deploy bridges on both networks and deploy AvaxToken, also update the admins"
).setAction(async (taskArgs, hre) => {
 await deploy().catch((error) => {
 console.error(error);
 process.exitCode = 1;
 });
});
...
...
```

Example [hardhat.config.js](#) file after adding the deploy task.

### **Run Deploy Script**

To run our deploy script, run:

```
npx hardhat deploy
```

After running the script you should see deployed contract addresses on the command line and see that `variables/contractAddresses.js` file is updated.

### **Balance Script**

#### **Write Balance Script**

Create `balance.js` file inside `scripts` folder

### [balance.js](#)

### **Update hardhat.config.js File**

To add `balance` script as a hardhat task add the following code inside `hardhat.config.js`

```
/* Import task from hardhat config if you did not already */
const { task } = require("hardhat/config");
...
/* Import balance function */
require("./scripts/balance");
...
/* Create balance task */
task("balance", "Get token balance from a network")
 /* Add `from` parameter indication the used network which is either avax or subnet */
 .addParam("from", "Network to get balance from")
 .setAction(async (taskArgs, hre) => {
 await balance(taskArgs.from).catch((error) => {
 console.error(error);
 process.exitCode = 1;
 });
 });
...
...
```

Example [hardhat.config.js](#) file after adding the balance task.

### **Run Balance Script**

To run our balance script, run:

```
npx hardhat balance --from avax
```

or

```
npx hardhat balance --from subnet
```

After running the script you should see balances printed out in the command line. If you have used `--from subnet` you will only see the native token balance of the user. If you have used `--from avax` you will see the ERC20 balances of both user and the bridge contract.

Balance value seen on the Subnet depends on how you have funded your address. But balance values on AVAX should look like the following:

```
MERC20 balance of the user: 1000000.0
MERC20 balance of the bridge: 0.0
```

### BurnOrLock Script

#### Write BurnOrLock Script

Create `burnOrLock.js` file inside `scripts` folder

[burnOrLock.js](#)

#### Update hardhat.config.js File

To add `burnOrLock` script as a hardhat task add the following code inside `hardhat.config.js`

```
/* Import task from hardhat config */
const { task } = require("hardhat/config");
...
/* Import burnOrLock function */
require("./scripts/burnOrLock");
...
/* Create burnOrRelease task */
task("burnOrLock", "Burn or lock token from a network")
 /* Add 'from' parameter indication the used network which is either avax or subnet */
 .addParam("from", "Network to burn or lock from")
 /* Add 'amount' parameter indication the amount to burn or lock */
 .addParam("amount", "Amount to burn or lock")
 .setAction(async (taskArgs, hre) => {
 await burnOrLock(taskArgs.from, taskArgs.amount).catch((error) => {
 console.error(error);
 process.exitCode = 1;
 });
 });
...

```

Example [hardhat.config.js](#) file after adding all tasks.

#### Run BurnOrRelease Script

To run `burnOrRelease` script, run:

```
npx hardhat burnOrLock --from avax --amount <amount-of-token-to-burn-or-lock-in-ether>
```

or

```
npx hardhat burnOrLock --from subnet --amount <Example value: 4>
```

:::caution

When you try to run the first script `... --from avax --amount 10` if the user has 10 ERC20 tokens it will work fine and you will see the updated balances as expected on AVAX network. User's decremented by 10, bridge's incremented by 10. But you would not see that the user's native token balance on the Subnet is increased. Although there are bridge contracts, there is no relayer application to establish the communication in between them. Therefore, the user locked its tokens but its balance on the Subnet did not change. It is the same for the second script where the user burns tokens on Subnet but does not get any new tokens on AVAX C-Chain. Be aware, if the user account does not have native token balance on the Subnet, the second script would throw an error.

:::

## Create the Relayer Application

Create a `relayer.js` file at root folder of the project.

[relayer.js](#)

#### Running the Relayer

As you can also see from the comments of the relayer file. There are different ways to start the relayer application

1. `node ./relayer.js`

- o When run with `node ./relayer.js` Relayer will subscribe to events from recent blocks on Avax and Subnet. Therefore, it might not process an event that is emitted 1000 blocks ago. If you want to start the relayer and make a transaction, current way of running is what you are looking for.

```
2. node ./relayer.js <avaxBlockNumber> <subnetBlockNumber>
```

- When run with `node ./relayer.js <avaxBlockNumber> <subnetBlockNumber>`. Relayer will look for events on Avax and Subnet from the block number you provided and will iterate through the next 10 blocks for the event. Therefore, if you have a burn or lock event emitted 1000 blocks ago, you can process it by giving the right blockNumber. If you want to start the relayer to process an old burn or lock event, current way of running is what you are looking for.

```
3. node ./relayer.js <avaxBlockNumber>
```

```
4. node ./relayer.js -1 <subnetBlockNumber>
```

- When run with `node ./relayer.js <avaxBlockNumber>` or `node ./relayer.js -1 <subnetBlockNumber>` Relayer will look for events on either Avax or Subnet from the block number you provided and will iterate through the next 10 blocks for the event. "-1" as block number means do not process any old blocks for that chain. Therefore, `node ./relayer.js -1 <subnetBlockNumber>` will only process events for the Subnet. If you want to start the relayer to process an old burn or lock event just on one chain, current way of running is what you are looking for

## Testing the Relayer

### Test Relayer

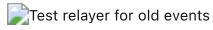
In this video on the left terminal I am using our custom scripts to interact with chains and on the right terminal I am using our relay to create the cross chain communication.



#### What Happens on the Video?

- Check balances on both chains
- Start the relayer
- An already processed event appears on the console of the relayer therefore it does not get processed
- Check the balances to make sure already processed event is not processed
- Lock 20 ERC20 token from AVAX and see the updated balances of user and the bridge on AVAX
- Relayer observes the transaction and sends a transaction to mint native tokens on Subnet
- Check the balances on both Subnet and AVAX. As expected our Subnet native token balance increases by 20
- Burn 5 native tokens from Subnet and see the updated balance of the user
- Relayer observes the transaction and sends a transaction to release 5 ERC20 tokens on AVAX
- Check balance on both Subnet and AVAX. As expected our ERC20 balance increases by 5 and bridge's decreases by 5

### Test Relayer for Old Events



#### What Happens on the Video?

- Start by a lock transaction from AVAX with amount 40. As stated in the video this transaction was sent when the relayer was not working. Therefore, it is not processed and it will not be processed when we start the relayer with `node relayer.js` because there has been many blocks after it
- Check balances on both chains
- Start the relayer with `node relayer.js` to show that the event is not getting processed. Printed events are events that happened on the Subnet. Since there are no blocks building on my local Subnet other than my own, my old burns are considered new and therefore shown
- Start the relayer with `node relayer.js <blockNumber>` to show that event will be processed and will be printed as "OLD: "
- Check balances on both chains to confirm that old lock event on AVAX has been processed by relayer and tokens have been minted on AVAX

## Troubleshoot Common Issues

Things to check out;

- Error while compiling contracts
  - You have run `npm i @openzeppelin/contracts`.
- Error while running scripts
  - Both accounts on both chains have some native token so that they can send transactions.
  - Folder structures and file names are as suggested. In our scripts we access the contract ABIs and bytecodes directly from the files that are created by hardhat. Those files are created according to your file structure and if you changed the structure, imports might fail.
  - You have your private keys inside the `.env` file and you have downloaded dotenv package by running `npm i dotenv`.
  - Your Subnet has NativeMinter precompile with `bridgeAdmin` account as the admin.
  - You have created a `contractAddresses.js` file inside the variables folder. If you did not create this file, `deploy.js` would fail.
  - If you are trying to bridge from Subnet to AVAX, in other words trying to burn from Subnet and release from AVAX, make sure that AvaxBridge has enough ERC20 tokens to release.

## Security and Maintenance

### Overview

Our bridge implementation is a naive one. It should give the reader a deep understanding of how bridges work rather than be used in production. There are many potential risks when running a bridge.

### Problems

- Bridge Operator Has so Much Power
  - Our bridge uses only one admin account to handle bridge funds. If that one private key gets hacked, hacker could easily steal all the funds.
- Smart Contracts
  - Contracts used in this tutorial are not audited.
- Relayer Application
  - There might be some bugs inside our relayer application.
  - We only have one relayer application running. Computer that runs the relayer application might encounter an error, maybe an electricity outage or we might need to update and restart the relayer application itself. In that case, our bridge would stop working for that time frame and users would not be able to bridge their tokens.

## Suggestions

- More Bridge Operators
  - Rather than trusting on a single account, we might use a multi-sig wallet. For example, it could be a 4 of 5 multi-sig wallet. Allowing us to confirm transactions with 5 accounts and on 4 confirmation it would allow for execution. In that case, we would have a multi-sig smart contract and that contract would be the admin of the bridges. There will be 5 different relayer applications running with 5 different accounts' private keys. Each of them would send transactions to the multi-sig smart contract. When 4 of them sends the transaction, smart contract would execute the mint or release function on the bridge contract. This approach would increase both the security and ease of maintenance of our bridge. Because of this approach, when one computer that runs the relayer application has to have a maintenance or have an electricity outage bridge would function normally. But, just because we have more bridge operators our bridge is not fully secure.
- Auditing Smart Contracts
  - Smart Contracts should be audited. But, do not forget audited does not mean it is bug free.
- Testing Relayer Application
  - Relayer application should be audited and tested as well.
  - If we decide to use a multi-sig wallet we have to update our relayer application to call relevant functions on the multi-sig contract rather than directly calling bridge contracts.

## Conclusion

Congratulations! You have created a bridge between AVAX C-Chain and your Subnet to use an ERC-20 token as a gas token.

Things achieved:

- Understood general design of bridges.
- Implemented bridge contracts.
- Used NativeMinter precompile.
- Created a relayer application to communicate with both chains.
- Tested relayer to make sure communication between chains are established.
- Used ERC-20 token as the gas token on the Subnet.

## References

- [Further Readings](#)

## Chains.js

```
module.exports = {
 avax: {
 chainId: 43113,
 rpcUrl: "https://api.avax-test.network/ext/bc/C/rpc",
 },
 subnet: {
 chainId: <your-subnet-chain-id>,
 rpcUrl:
 "<your-subnet-rpc-url>",
 },
};
```

## NativeMinterAddress.js

```
module.exports = {
 SUBNET_NATIVE_MINTER_ADDRESS: "0x020001",
};
```

## AvaxBridge.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```

contract AvaxBridge {
 address public admin;
 /* Gets incremented with each `lock()`, indicates the transferCount
 and prevents double processing the event */
 uint public nonce;

 /* Represents the ERC20 token */
 IERC20 public avaxToken;

 /* Mapping to hold whether nonce is processed or not */
 mapping(uint => bool) public processedNonces;

 /* Allows us to indicate whether it is a `release()` or `lock()` when emitting an event */
 enum Type {
 Release,
 Lock
 }

 /*
 * Event that is emitted with both `release()` and `lock()`
 * Relayer listens to events emitted by `lock()`
 * Potential frontend application may want to listen to events emitted by `release()`
 */
 event Transfer(
 address from,
 address to,
 uint amount,
 uint time,
 uint nonce,
 Type indexed transferType
);

 /* Modifier to allow some functions to be only called by admin */
 modifier onlyAdmin() {
 require(msg.sender == admin, "only admin");
 _;
 }

 /* Constructor that sets admin as the sender and initializes the ERC20 token inside contract */
 constructor(address _token) {
 admin = msg.sender;
 avaxToken = IERC20(_token);
 }

 /* Function to allow setting a new admin */
 function setAdmin(address newAdmin) external onlyAdmin {
 admin = newAdmin;
 }

 /* Function that is called by the relayer to release some tokens after it is burned on the subnet */
 function release(
 address to,
 uint amount,
 uint subnetNonce
) external onlyAdmin {
 require(
 processedNonces[subnetNonce] == false,
 "nonce already processed"
);
 processedNonces[subnetNonce] = true;

 /* Bridge sends locked tokens to the `to` address therefore, releases the tokens */
 avaxToken.transfer(to, amount);

 emit Transfer(
 msg.sender,
 to,
 amount,
 block.timestamp,
 subnetNonce,
 Type.Release
);
 }

 /*
 * Function that is called by the user to lock their tokens.
 * Relayer listens to the event emitted by this function and if the nonce is not processed,
 * it will call `mint()` of the SubnetBridge
 */
}

```

```

*/
function lock(address to, uint amount) external {
 /* Send ERC20 tokens from msg.sender (user) to bridge to lock the tokens */
 /* Do not forget: sender should approve bridge address to do this */
 avaxToken.transferFrom(msg.sender, address(this), amount);

 /* Event that is emitted for relayer to process */
 emit Transfer(
 msg.sender,
 to,
 amount,
 block.timestamp,
 nonce,
 Type.Lock
);
 /* Increment the nonce to prevent double counting */
 nonce++;
}
}

```

## SubnetBridge.sol

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "../Token/INativeMinter.sol";

contract SubnetBridge {
 address public admin;
 /* Address to send tokens to burn them */
 address public burnAddress = address(0x0);
 /* Gets incremented with each `burn()`, indicates the transferCount
 and prevents double processing the event */
 uint public nonce;

 /* Represents NativeMinterInterface */
 NativeMinterInterface public nativeMinter =
 NativeMinterInterface(
 /*
 Native Minter contract is always at this address
 as explained at https://docs.avax.network/subnets/customize-a-subnet#minting-native-coins
 */
 address(0x0200)
);

 /* Mapping to hold whether nonce is processed or not */
 mapping(uint => bool) public processedNonces;

 /* Allows us to indicate whether it is a `mint()` or `burn()` when emitting an event */
 enum Type {
 Mint,
 Burn
 }

 /*
 Event that is emitted with both `mint()` and `burn()`
 Relayer listens to events emitted by `burn()`
 Potential frontend application may want to listen to events emitted by `mint()`
 */
 event Transfer(
 address from,
 address to,
 uint amount,
 uint time,
 uint nonce,
 Type indexed transferType
);

 /* Modifier to allow some functions to be only called by admin */
 modifier onlyAdmin() {
 require(msg.sender == admin, "only admin");
 _;
 }

 /* Constructor that sets admin as the sender */

```

```

constructor() {
 admin = msg.sender;
}

/* Function to allow setting new admin */
function setAdmin(address newAdmin) external onlyAdmin {
 admin = newAdmin;
}

/* Function that is called by the relayer to mint some tokens after it is locked on the avax */
function mint(
 address to,
 uint amount,
 uint avaxNonce
) external onlyAdmin {
 require(processedNonces[avaxNonce] == false, "nonce already processed");
 processedNonces[avaxNonce] = true;

 nativeMinter.mintNativeCoin(to, amount);
 emit Transfer(
 msg.sender,
 to,
 amount,
 block.timestamp,
 avaxNonce,
 Type.Mint
);
}

/*
Function that is called by the user to burn their tokens.
Relayer listens to this event and if the nonce is not processed,
it will call `release()` of the AvaxBridge
*/
function burn(address to) external payable {
 require(msg.value > 0, "You have to burn more than 0 tokens");
 /* Send native token to 0x0 address, effectively burning native token */
 (bool sent,) = payable(burnAddress).call{value: msg.value}("");
 require(sent, "Failed to send native token");

 /* Event that is emitted for relayer to process */
 emit Transfer(
 msg.sender,
 to,
 msg.value,
 block.timestamp,
 nonce,
 Type.Burn
);

 /* Increment the nonce to prevent double counting */
 nonce++;
}
}

```

## AvaxToken.sol

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

// A standard ERC20 token with maxSupply of 1 million
contract AvaxToken is ERC20 {
 uint public MAX_SUPPLY = 1000000 ether;

 // maxSupply is sent to the creator of the token
 constructor(string memory name, string memory symbol) ERC20(name, symbol) {
 _mint(msg.sender, MAX_SUPPLY);
 }
}

```

## INativeMinter.sol

```

// (c) 2022-2023, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

interface NativeMinterInterface {
 // Set [addr] to have the admin role over the minter list
 function setAdmin(address addr) external;

 // Set [addr] to be enabled on the minter list
 function setEnabled(address addr) external;

 // Set [addr] to have no role over the minter list
 function setNone(address addr) external;

 // Read the status of [addr]
 function readAllowList(address addr) external view returns (uint256);

 // Mint [amount] number of native coins and send to [addr]
 function mintNativeCoin(address addr, uint256 amount) external;
}

```

## HardhatConfigAfterBalance.js

```

require("@nomicfoundation/hardhat-toolbox")
/* Import task from hardhat config */
const { task } = require("hardhat/config")

/* Import deploy function */
require("./scripts/deploy")
/* Import balance function */
require("./scripts/balance")

/* Create deploy task */
task(
 "deploy",
 "Deploy bridges on both networks and deploy AvaxToken, also update the admins"
).setAction(async (taskArgs, hre) => {
 await deploy().catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/* Create balance task */
task("balance", "Get token balance from a network")
/* Add `from` parameter indication the used network which is either avax or subnet */
.addParam("from", "Network to get balance from")
.setAction(async (taskArgs, hre) => {
 await balance(taskArgs.from).catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
 solidity: "0.8.9",
}

```

## HardhatConfigAfterBurnOrLock.js

```

require("@nomicfoundation/hardhat-toolbox")
/* Import task from hardhat config */
const { task } = require("hardhat/config")

/* Import deploy function */
require("./scripts/deploy")
/* Import balance function */
require("./scripts/balance")

```

```

/* Import burnOrLock function */
require("./scripts/burnOrLock")

/* Create deploy task */
task(
 "deploy",
 "Deploy bridges on both networks and deploy AvaxToken, also update the admins"
).setAction(async (taskArgs, hre) => {
 await deploy().catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/* Create balance task */
task("balance", "Get token balance from a network")
/* Add `from` parameter indication the used network which is either avax or subnet */
.addParam("from", "Network to get balance from")
.setAction(async (taskArgs, hre) => {
 await balance(taskArgs.from).catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/* Create burnOrRelease task */
task("burnOrLock", "Burn or lock token from a network")
/* Add `from` parameter indication the used network which is either avax or subnet */
.addParam("from", "Network to burn or lock from")
/* Add `amount` parameter indication the amount to burn or lock */
.addParam("amount", "Amount to burn or lock")
.setAction(async (taskArgs, hre) => {
 await burnOrLock(taskArgs.from, taskArgs.amount).catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
 solidity: "0.8.9",
}

```

## HardhatConfigAfterDeploy.js

```

require("@nomicfoundation/hardhat-toolbox")
/* Import task from hardhat config */
const { task } = require("hardhat/config")

/* Import deploy function */
require("./scripts/deploy")

/* Create deploy task */
task(
 "deploy",
 "Deploy bridges on both networks and deploy AvaxToken, also update the admins"
).setAction(async (taskArgs, hre) => {
 await deploy().catch((error) => {
 console.error(error)
 process.exitCode = 1
 })
})

/** @type import('hardhat/config').HardhatUserConfig */
module.exports = {
 solidity: "0.8.9",
}

```

## relayer.js

```

const { ethers } = require("ethers");
const dotenv = require("dotenv");

```

```

/* Get needed util functions */
const initProviders = require("./utils/initProviders");
const initSigners = require("./utils/initSigners");
const initContracts = require("./utils/initContracts");

dotenv.config();

/* Relayer application

Could be run by
`node ./relayer.js`,
`node ./relayer.js <avaxBlockNumber>`,
`node ./relayer.js <avaxBlockNumber> <subnetBlockNumber>`,
`node ./relayer.js -1 <subnetBlockNumber>`

When run with `node ./relayer.js`:
Relayer will subscribe to events from recent blocks on Avax and Subnet
Therefore, it might not process an event that is emitted 1000 blocks ago
If you want to start the relayer and make a transaction, current way of running is what you are looking for

When run with `node ./relayer.js <avaxBlockNumber> <subnetBlockNumber>`:
Relayer will look for events on Avax and Subnet from the block number you provided
and will iterate through the next 10 blocks for the event. Will process observed event
Therefore, if you have a burn or lock event emitted 1000 blocks ago, you can process it by giving the right blockNumber
If you want to start the relayer to process an old burn or lock event, current way of running is what you are looking for

When run with `node ./relayer.js -1 <subnetBlockNumber>` or `node ./relayer.js <avaxBlockNumber>`:
Relayer will look for events on either Avax or Subnet from the block number you provided
and will iterate through the next 10 blocks for the event. Will process observed event
"-1" as block number means do not process any old blocks for that chain.
Therefore, `node ./relayer.js -1 <subnetBlockNumber>` will only process events for the subnet.
If you want to start the relayer to process an old burn or lock event just on one chain, current way of running is what
you are looking for
*/
const main = async () => {
/*
 If there is a need for sending transactions
 Add it to the txs array.
 Because of the `setInterval()`` relayer will send transactions every 5 seconds if at least 1 transaction exists
 We wait 5 seconds in between transactions to make sure we do not replace our own transactions before they are added to
a block.
*/
let txs = [];

/* Init providers, signers and bridgeContracts */
const providers = initProviders();
const signers = initSigners(providers);
const bridgeContracts = initContracts(signers);

/*
For Avax
Relayer gets command line arguments
If command line argument exists and it is not -1
Then process next 10 blocks and process events from these block number
*/
if (process.argv[2] && parseInt(process.argv[2]) !== -1) {
 const startBlock = parseInt(process.argv[2]);
 const recentBlock = await providers.avax.getBlockNumber();
 /*
 If startBlock + 10 exceeds the recent block it would throw an error.
 Since we would be trying to process blocks that are not there.
 Therefore, we set endBlock to the smaller one of two
 */
 const endBlock =
 startBlock + 10 >= recentBlock ? recentBlock : startBlock + 10;
 /* Reset blockNumber of the provider to process those blocks */
 providers.avax.resetEventsBlock(startBlock);
 /* Create new bridge contract instance because provider is changed */
 const localBridgeContracts = initContracts(signers);
 /* Filter events for "Transfer" event */
 const filter = localBridgeContracts.avax.admin.filters.Transfer();
 /* Query contract for old events in between startBlock and endBlock with given filter */
 let oldAvaxEvents = await localBridgeContracts.avax.admin.queryFilter(
 filter,
 startBlock,
 endBlock
);
 /* Format oldEvents as if they are transactions */
}

```

```

oldAvaxEvents = oldAvaxEvents.map((event) => ({
 chain: "subnet",
 to: event.args.to,
 amount: event.args.amount,
 nonce: event.args.nonce,
}));

/*
 Since we are looking for old events
 They might have been already processed
 Therefore, check if corresponding nonce is already processed or not
 If not, add it to the txs array
*/
await Promise.all(
 oldAvaxEvents.map(async (event) => {
 const { to, amount, nonce } = event;
 console.log("OLD: Lock happened on avax");
 console.log(
 `OLD: Transfer: to: ${to}, amount: ${ethers.utils.formatEther(
 amount
)}, nonce: ${nonce}`
);
 /* Check if nonce is processed or not */
 const isProcessed =
 await localBridgeContracts.subnet.admin.processedNonces(nonce);
 if (!isProcessed) {
 /* If not processed add tx to txs array */
 console.log("OLD: is not processed, will mint on subnet\n");
 txs.push(event);
 } else {
 console.log("OLD: is already processed\n");
 }
 })
);
}

// Pretty familiar as above, provider is changed
/*
 For Subnet
 Relayer gets command line arguments
 If command line argument exists and it is not -1
 Then process next 10 blocks and process events from these block number
*/
if (process.argv[3] && parseInt(process.argv[3]) !== -1) {
 const startBlock = parseInt(process.argv[3]);
 const recentBlock = await providers.subnet.getBlockNumber();
 /*
 If startBlock + 10 exceeds the recent block it would throw an error.
 Since we would be trying to process blocks that are not there.
 Therefore, we set endBlock to the smaller one of two
 */
 const endBlock =
 startBlock + 10 >= recentBlock ? recentBlock : startBlock + 10;
 /* Reset blockNumber of the provider to process those blocks */
 providers.subnet.resetEventsBlock(startBlock);
 /* Create new bridge contract instance because provider is changed */
 const localBridgeContracts = initContracts(signers);
 /* Filter events for "Transfer" event */
 const filter = localBridgeContracts.subnet.admin.filters.Transfer();
 /* Query contract for old events in between startBlock and endBlock with given filter */
 let oldSubnetEvents = await localBridgeContracts.subnet.admin.queryFilter(
 filter,
 startBlock,
 endBlock
);
 /* Format oldEvents as if they are transactions */
 oldSubnetEvents = oldSubnetEvents.map((event) => ({
 chain: "avax",
 to: event.args.to,
 amount: event.args.amount,
 nonce: event.args.nonce,
})));
}

/*
 Since we are looking for old events
 They might have been already processed
 Therefore, check if corresponding nonce is already processed or not
 If not, add it to the txs array
*/

```

```

 */
 await Promise.all(
 oldSubnetEvents.map(async (event) => {
 const { to, amount, nonce } = event;
 console.log("OLD: Burned happened on subnet");
 console.log(
 `OLD: Transfer: to: ${to}, amount: ${ethers.utils.formatEther(
 amount
)}, nonce: ${nonce}`
);
 /* Check if nonce is processed or not */
 const isProcessed =
 await localBridgeContracts.avax.admin.processedNonces(nonce);
 if (!isProcessed) {
 /* If not processed add tx to txs array */
 console.log("OLD: is not processed, will release on subnet\n");
 txs.push(event);
 } else {
 console.log("OLD: is already processed\n");
 }
 })
);
 }

 /* With above 2 functions we have processed old blocks */
 console.log("\n\nOld events processed");

 /*
 Now we subscribe to bridgeContract events on both chains
 Which allows us to run a function whenever a new event is observed
 */

 /* Subscribe to bridge events on avax */
 bridgeContracts.avax.admin.on(
 /* Subscribe to "Transfer" event */
 "Transfer",
 async (from, to, amount, date, nonce, type) => {
 /*
 type 0 means it is a release event
 type 1 means it is a lock event

 We only care for lock events as relayer. On lock events we will mint on subnet
 We have added the release event for frontend applications.
 */
 if (type === 1) {
 console.log("Lock happened on avax");
 console.log(
 `Transfer: from: ${from}, to: ${to}, amount: ${ethers.utils.formatEther(
 amount
)}, date: ${date}, nonce: ${nonce}, type: ${type}`
);
 } /* Check if nonce is processed or not */
 try {
 const isProcessed = await bridgeContracts.subnet.admin.processedNonces(
 nonce
);
 if (!isProcessed) {
 /* If not processed add tx to txs array */
 console.log("is not processed, will mint on subnet\n");
 txs.push({ chain: "subnet", to, amount, nonce });
 } else {
 console.log("is already processed\n");
 }
 } catch (error) {
 console.log("error while checking processedNonces on subnet bridge: ", error);
 }
 }
);
}

/* Subscribe to bridge events on subnet */
bridgeContracts.subnet.admin.on(
 /* Subscribe to "Transfer" event */
 "Transfer",
 async (from, to, amount, date, nonce, type) => {
 /*
 type 0 means it is a mint event
 type 1 means it is a burn event
 */

```

```

 We only care for burn events as relayer. On burn events we will release on subnet
 We have added the mint event for frontend applications.
 */
if (type === 1) {
 console.log("Burn happened on subnet");
 console.log(
 `Transfer: from: ${from}, to: ${to}, amount: ${ethers.utils.formatEther(
 amount
)}, date: ${date}, nonce: ${nonce}, type: ${type}`
);
 /* Check if nonce is processed or not */
 try {
 const isProcessed = await bridgeContracts.avax.admin.processedNonces(
 nonce
);
 if (!isProcessed) {
 /* If not processed add tx to txs array */
 console.log("is not processed, will release on avax\n");
 txs.push({ chain: "avax", to, amount, nonce });
 } else {
 console.log("is already processed\n");
 }
 } catch {
 console.log("error while checking processedNonces on avax bridge: ", error);
 }
}
};

console.log("Started listening for new events\n\n");

/*
 This function gets to run each 5 seconds and it sends `mint` or `release` transactions to the bridge contract
 We wait 5 seconds in between transactions to make sure we do not replace our own transactions before they are added to a
block.
*/
setInterval(async () => {
 /* If there is no transaction to send, do nothing */
 if (txs.length > 0) {
 /*
 If provided blockNumbers for avax or subnet are close to current blocks of the chains
 Then a transaction might get added to the txs array twice. Once processing old blocks (but pretty recent) and
 once subscribed to new events. Therefore, we have to eliminate same txs by filtering.
 */
 txs = txs.filter((value, index) => {
 const _value = JSON.stringify(value);
 return index === txs.findIndex(obj => {
 return JSON.stringify(obj) === _value;
 })
 });
 console.log("txs: ", txs);
 let tx;
 /* Remove the first element from the array and destructure it */
 const { chain, to, amount, nonce } = txs.shift();
 /* Check which chain the transaction will be sent to */
 try {
 if (chain === "avax") {
 /* Call `release()` on avax */
 tx = await bridgeContracts[chain].admin.release(to, amount, nonce);
 } else if (chain === "subnet") {
 /* Call `mint()` on subnet */
 tx = await bridgeContracts[chain].admin.mint(to, amount, nonce);
 } else return;
 await tx.wait();
 console.log("transaction processed, token minted or released");
 } catch (error) {
 console.log("error sending transaction: ", error);
 }
 }
 , 5000);
};

main().catch((error) => {
 console.error(error);
 process.exitCode = 1;
});

```

## Balance.js

```
const { ethers } = require("ethers");
const dotenv = require("dotenv");

const chains = require("../constants/chains");
/* Get contract addresses from the file we generated by running the deploy script */
const {
 AVAX_TOKEN_ADDRESS,
 AVAX_BRIDGE_ADDRESS,
} = require("../variables/contractAddresses");
/* Get ABIs of the contracts directly from the artifact folder created by hardhat after each compilation */
const AVAX_TOKEN_ABI =
 require("../artifacts/contracts	TokenName/AvaxToken.sol/AvaxToken").abi;
dotenv.config();

/*
 Balance script that allows us to check balances on both chains
 On Avax it prints out the ERC20 balance of the user and the bridge.
 (We are printing out the balance of the bridge because as users lock tokens bridge's balance of the ERC20 will increase)
 On Subnet it prints out the native token balance of the user
*/
module.exports = balance = async (from) => {
 let provider;
 let signer;
 let contract;
 /* This script takes command line argument to indicate which chain we are using */
 if (from === "avax") {
 /* Initialize; provider, signer and contract */
 provider = new ethers.providers.JsonRpcProvider(chains.avax.rpcUrl);
 signer = new ethers.Wallet(process.env.BRIDGE_USER_PRIVATE_KEY, provider);
 contract = new ethers.Contract(AVAX_TOKEN_ADDRESS, AVAX_TOKEN_ABI, signer);
 /* Get ERC20 balance of the user */
 const newUserBalance = await contract.balanceOf(signer.address);
 /* Get ERC20 balance of the bridge */
 const newBridgeBalance = await contract.balanceOf(AVAX_BRIDGE_ADDRESS);
 console.log(
 "ERC20 balance of the user: ",
 ethers.utils.formatEther(newUserBalance)
);
 console.log(
 "ERC20 balance of the bridge: ",
 ethers.utils.formatEther(newBridgeBalance)
);
 } else if (from === "subnet") {
 /* Initialize; provider, signer */
 provider = new ethers.providers.JsonRpcProvider(chains.subnet.rpcUrl);
 signer = new ethers.Wallet(process.env.BRIDGE_USER_PRIVATE_KEY, provider);
 /* Get native token balance of the user */
 const balance = await signer.getBalance();
 console.log(
 "Native token balance on Subnet: ",
 ethers.utils.formatEther(balance)
);
 } else {
 return;
 }
};
```

## BurnOrLock.js

```
const { ethers } = require("ethers");
const dotenv = require("dotenv");

const chains = require("../constants/chains");
/* Get contract addresses from the file we generated by running the deploy script */
const {
 AVAX_BRIDGE_ADDRESS,
 AVAX_TOKEN_ADDRESS,
 SUBNET_BRIDGE_ADDRESS,
} = require("../variables/contractAddresses");
/* Get ABIs of the contracts directly from the artifact folder created by hardhat after each compilation */
const SUBNET_BRIDGE_ABI =
```

```

require("../artifacts/contracts/Bridge/SubnetBridge.sol/SubnetBridge").abi;
const AVAX_BRIDGE_ABI =
 require("../artifacts/contracts/Bridge/AvaxBridge.sol/AvaxBridge").abi;
const AVAX_TOKEN_ABI =
 require("../artifacts/contracts	TokenName.sol/AvaxToken").abi;
dotenv.config();

/*
BurnOrLock script that allows us to both burn and lock tokens
On Avax it allows us to lock ERC20 tokens
On Subnet it allows us to burn native tokens
*/
module.exports = burnOrLock = async (from, amount) => {
 let provider;
 let signer;
 let bridgeContract;
 /* This script takes command line argument to indicate which chain we are using */
 if (from === "avax") {
 /* Initialize; provider, signer and tokenContract */
 provider = new ethers.providers.JsonRpcProvider(chains.avax.rpcUrl);
 signer = new ethers.Wallet(process.env.BRIDGE_USER_PRIVATE_KEY, provider);
 const tokenContract = new ethers.Contract(
 AVAX_TOKEN_ADDRESS,
 AVAX_TOKEN_ABI,
 signer
);

 /* Approve bridge to use the token of the sender before trying to lock tokens */
 const approveTx = await tokenContract.approve(
 AVAX_BRIDGE_ADDRESS,
 ethers.utils.parseEther(amount)
);
 await approveTx.wait();

 /* Initialize bridgeContract */
 bridgeContract = new ethers.Contract(
 AVAX_BRIDGE_ADDRESS,
 AVAX_BRIDGE_ABI,
 signer
);

 /* User locks ERC20 tokens to the bridge */
 const lockTx = await bridgeContract.lock(
 signer.address,
 ethers.utils.parseEther(amount)
);
 const minedTx = await lockTx.wait();

 console.log("Successfully locked amount on avax: ", amount);
 console.log("At block: ", minedTx.blockNumber);
 /* Get user's ERC20 balance after lock */
 const newUserBalance = await tokenContract.balanceOf(signer.address);
 /* Get bridge's ERC20 balance after lock */
 const newBridgeBalance = await tokenContract.balanceOf(AVAX_BRIDGE_ADDRESS);
 console.log(
 "Updated balance of user after lock: ",
 ethers.utils.formatEther(newUserBalance)
);
 console.log(
 "Updated balance of bridge after lock: ",
 ethers.utils.formatEther(newBridgeBalance)
);
 } else if (from === "subnet") {
 /* Initialize; provider, signer and tokenContract */
 provider = new ethers.providers.JsonRpcProvider(chains.subnet.rpcUrl);
 signer = new ethers.Wallet(process.env.BRIDGE_USER_PRIVATE_KEY, provider);
 bridgeContract = new ethers.Contract(
 SUBNET_BRIDGE_ADDRESS,
 SUBNET_BRIDGE_ABI,
 signer
);

 /* User burns native tokens */
 const burnTx = await bridgeContract.burn(signer.address, {
 value: ethers.utils.parseEther(amount),
 });
 const minedTx = await burnTx.wait();
 console.log("Successfully burned amount on subnet: ", amount);
 console.log("At block: ", minedTx.blockNumber);
 }
}

```

```

 /* Get user's native token balance after burn */
 const newUserBalance = await signer.getBalance();
 console.log(
 "Updated balance of user after burn: ",
 ethers.utils.formatEther(newUserBalance)
);
 } else {
 return;
 }
};

}

```

## Deploy.js

```

const fs = require("fs");
const { ethers } = require("ethers");
const dotenv = require("dotenv");

/* Get NativeMinter address from constants */
const {
 SUBNET_NATIVE_MINTER_ADDRESS,
} = require("../constants/nativeMinterAddress");

/* Get ABIs of the contracts directly from the artifact folder created by hardhat after each compilation */
/* Get bytecodes of the contracts directly from the artifact folder created by hardhat after each compilation */
const AVAX_TOKEN_BYTECODE =
 require("../artifacts/contracts/Token/AvaxToken.sol/AvaxToken").bytecode;
const AVAX_TOKEN_ABI =
 require("../artifacts/contracts/Token/AvaxToken.sol/AvaxToken").abi;
const AVAX_BRIDGE_BYTECODE =
 require("../artifacts/contracts/Bridge/AvaxBridge.sol/AvaxBridge").bytecode;
const AVAX_BRIDGE_ABI =
 require("../artifacts/contracts/Bridge/AvaxBridge.sol/AvaxBridge").abi;
const SUBNET_BRIDGE_BYTECODE =
 require("../artifacts/contracts/Bridge/SubnetBridge.sol/SubnetBridge").bytecode;
const SUBNET_BRIDGE_ABI =
 require("../artifacts/contracts/Bridge/SubnetBridge.sol/SubnetBridge").abi;
const SUBNET_NATIVE_MINTER_ABI =
 require("../artifacts/contracts/Token/INativeMinter.sol/NativeMinterInterface").abi;

/* Get needed util functions */
const initProviders = require("../utils/initProviders");
const initSigners = require("../utils/initSigners");
dotenv.config();

/*
Deploy script that allows us to deploy:
 AvaxToken, AvaxBridge, SubnetBridge contracts
 And set SubnetBridge contract as a `Minter` for the NativeMinter precompile
*/
module.exports = deploy = async () => {
 const providers = initProviders();
 const signers = initSigners(providers);

 /*
 Deploy AvaxToken it gives the total supply of the token to the msg.sender
 (which is the user that will use the bridge in our case)
 */
 const AvaxTokenFactory = new ethers.ContractFactory(
 AVAX_TOKEN_ABI,
 AVAX_TOKEN_BYTECODE,
 signers.avax.user
);
 const avaxToken = await AvaxTokenFactory.deploy("MyErc20", "MERC20");
 await avaxToken.deployTransaction.wait();
 console.log("avax token deployed to: ", avaxToken.address);

 /* Deploy AvaxBridge it makes msg.sender the admin of the bridge */
 const AvaxBridgeFactory = new ethers.ContractFactory(
 AVAX_BRIDGE_ABI,
 AVAX_BRIDGE_BYTECODE,
 signers.avax.admin
);
 const avaxBridge = await AvaxBridgeFactory.deploy(avaxToken.address);
 await avaxBridge.deployTransaction.wait();
}

```

```

console.log("avax bridge deployed to: ", avaxBridge.address);

/* Deploy SubnetBridge it makes msg.sender the admin of the bridge */
const SubnetBridgeFactory = new ethers.ContractFactory(
 SUBNET_BRIDGE_ABI,
 SUBNET_BRIDGE_BYTECODE,
 signers.subnet.admin
);
const subnetBridge = await SubnetBridgeFactory.deploy();
await subnetBridge.deployTransaction.wait();
console.log("subnet bridge deployed to: ", subnetBridge.address);

/* Give 'Minter' role to SubnetBridge contract so that it can mint native token */
const nativeMinter = new ethers.Contract(
 SUBNET_NATIVE_MINTER_ADDRESS,
 SUBNET_NATIVE_MINTER_ABI,
 signers.subnet.admin
);
const setNativeMinterTx = await nativeMinter.setEnabled(subnetBridge.address);
await setNativeMinterTx.wait();
console.log("allowed subnet bridge to mint native coins");

/*
Whenever we run this deploy script, deployed contract addresses will be changed.
Rather than manually updating them we write the updated address to the `variables/contractAddress.js`
Inside our code, whenever we try to access the address of a contract we use this file as the source of truth.
*/
fs.writeFileSync(
 "variables/contractAddresses.js",
 `module.exports = {
 AVAX_TOKEN_ADDRESS: "${avaxToken.address}",
 AVAX_BRIDGE_ADDRESS: "${avaxBridge.address}",
 SUBNET_BRIDGE_ADDRESS: "${subnetBridge.address}",
 }`,
);
console.log(
 "updated contract addresses inside variables/contractAddresses.js"
);
};

}

```

## InitContracts.js

```

const { ethers } = require("ethers");
/* Get ABIs of the contracts directly from the artifact folder created by hardhat after each compilation */
const SUBNET_BRIDGE_ABI =
 require("../artifacts/contracts/Bridge/SubnetBridge.sol/SubnetBridge").abi;
const AVAX_BRIDGE_ABI =
 require("../artifacts/contracts/Bridge/AvaxBridge.sol/AvaxBridge").abi;
/* Currently we did not deploy our contracts but when we deploy them we will store them in the following file */
const {
 AVAX_BRIDGE_ADDRESS,
 SUBNET_BRIDGE_ADDRESS,
} = require("../variables/contractAddresses");

module.exports = (signers) => {
 /* AvaxBridge contract with signer access of bridgeAdmin */
 const avaxBridgeAdmin = new ethers.Contract(
 AVAX_BRIDGE_ADDRESS,
 AVAX_BRIDGE_ABI,
 signers.avax.admin
);
 /* AvaxBridge contract with signer access of user */
 const avaxBridgeUser = new ethers.Contract(
 AVAX_BRIDGE_ADDRESS,
 AVAX_BRIDGE_ABI,
 signers.avax.user
);
 /* SubnetBridge contract with signer access of bridgeAdmin */
 const subnetBridgeAdmin = new ethers.Contract(
 SUBNET_BRIDGE_ADDRESS,
 SUBNET_BRIDGE_ABI,
 signers.subnet.admin
);
 /* SubnetBridge contract with signer access of user */
 const subnetBridgeUser = new ethers.Contract(

```

```

 SUBNET_BRIDGE_ADDRESS,
 SUBNET_BRIDGE_ABI,
 signers.subnet.user
);

 return {
 avax: { admin: avaxBridgeAdmin, user: avaxBridgeUser },
 subnet: { admin: subnetBridgeAdmin, user: subnetBridgeUser },
 };
};

}

```

## InitProviders.js

```

const { ethers } = require("ethers");
const chains = require("../constants/chains");

module.exports = () => {
 /* Create providers for both chains */
 const avaxProvider = new ethers.providers.JsonRpcProvider(chains.avax.rpcUrl);
 const subnetProvider = new ethers.providers.JsonRpcProvider(
 chains.subnet.rpcUrl
);
 return { avax: avaxProvider, subnet: subnetProvider };
};

```

## InitSigners.js

```

const { ethers } = require("ethers")
const dotenv = require("dotenv")
dotenv.config()

module.exports = (providers) => {
 /*
 Since we will have a bridge admin account to deploy and interact with bridges
 and a user account that is using the bridge.
 We have 2 different accounts to interact with bridges on 2 different chains
 Therefore, we have to create 4 wallets
 */
 const avaxBridgeAdmin = new ethers.Wallet(
 process.env.BRIDGE_ADMIN_PRIVATE_KEY,
 providers.avax
)
 const avaxBridgeUser = new ethers.Wallet(
 process.env.BRIDGE_USER_PRIVATE_KEY,
 providers.avax
)
 const subnetBridgeAdmin = new ethers.Wallet(
 process.env.BRIDGE_ADMIN_PRIVATE_KEY,
 providers.subnet
)
 const subnetBridgeUser = new ethers.Wallet(
 process.env.BRIDGE_USER_PRIVATE_KEY,
 providers.subnet
)
 return {
 avax: { admin: avaxBridgeAdmin, user: avaxBridgeUser },
 subnet: { admin: subnetBridgeAdmin, user: subnetBridgeUser },
 }
}

```

:::info

As you see, we have used `process.env.<..._PRIVATE_KEY>`. Reason behind that is we do not want to expose our private keys inside our code. To install the related package run:

```
npm i dotenv
```

Then, at the root of the project create a file named `.env`. Afterwards put in the private keys of your accounts as follows:

```
BRIDGE_ADMIN_PRIVATE_KEY=<private-key-for-admin>
BRIDGE_USER_PRIVATE_KEY=<private-key-for-user>
```

- Make sure that both accounts are funded on both chains so that they can send transactions.
- Make sure that `.env` file is included in your `.gitignore` file so that you do not upload this file to git.

:::

## Set up Your Development Environment for Local Subnet Development

If you are not familiar with Subnets, virtual machines or similar terminology you can refer to [Subnet Overview](#).

### Introduction

This tutorial's goal is to deploy and start a basic Subnet in your local machine. So that you can interact with the Subnet using [Remix](#) and [Hardhat](#). In this tutorial we will be using [avalanche-cli](#) to create and deploy the Subnet. If you ever encounter an error, refer to [Troubleshoot Common Issues](#) section.

:::info

avalanche-cli is in beta version. So it might get updated fairly frequently. It is best to refer to the latest version from its [github page](#).

:::

If you want to customize your Subnet you can refer to the optional [Customize the Subnet](#) section, while creating your Subnet.

Steps to follow:

1. Install avalanche-cli
2. Create the Subnet
3. Deploy the Subnet
4. Interact with the Subnet
  - 4.1 Using Remix
  - 4.2 Using Hardhat
5. Interact with Precompiles (Optional)

### Requirements

- Mac or Linux environment

### Step 1: Install Avalanche-Cli

To build avalanche-cli you have to first install Golang. Follow the instructions here: <https://go.dev/doc/install>.

After downloading Golang, to download avalanche-cli's latest version, run:

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s
```

:::note

This command will download the bin to the `./` (relative to where the command has run). To download in a custom location refer to [Installing in Custom Location](#).

:::

To add avalanche to PATH, run:

```
cd bin; \
export PATH=$PWD:$PATH
```

:::note

This command will add avalanche-cli to the PATH temporarily, which means that, when you reopen your terminal you would not be able to run 'avalanche' command. So, to add it permanently refer to [Add Avalanche Command Permanently](#) section.

:::

### Installing in Custom Location

To download the binary to a specific directory, run:

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s -- -b <relative-directory>
```

### Add Avalanche Command Permanently

To add avalanche command to your path:

#### MacOS

1. Open shell config file.
  - o If you are using `zsh` shell, open `~/.zprofile` in a text editor.

- o If you are suing bash shell, open `$(HOME)/.bash_profile` in a text editor.

2. Add this line, `export PATH=<path-of-avalanche-bin-directory>:$PATH` to the file.

3. Restart the terminal and run `avalanche`.

Linux

1. Open shell config file, `~/.bashrc` in a text editor.
  2. Add this line, `export PATH=<path-of-avalanche-bin-directory>:$PATH` to the file.
  3. Restart the terminal and run `avalanche`.

**:::note**

If you have run the installation command at `$HOME` directory, `<path-of-avalanche-bin-directory>` would be  `${HOME}/bin`.

• • •

## Step 2: Create the Subnet

To create the Subnet, run:

```
avalanche subnet create <subnetName>
```

When you run this command you will be walked through the customization of the Subnet. You can learn more about the configuration details at [Customize the Subnet](#) section.

## Example walk through:

- Choose your VM : SubnetEVM
  - ChainId : 676767
  - Token symbol : SUB
  - How would you like to set the fees : Low disk use...
  - How would you like to distribute the funds : Airdrop 1 million tokens to the default address
  - Advanced: Would you like to add a custom precompile to modify the EVM? : No

You have successfully created the genesis file for your Subnet. You can read more about genesis [here](#).

To see details about the Subnet, run:

```
avalanche subnet describe <subnetName>
```

To see the genesis file directly, run:

```
avalanche subnet describe <subnetName> --genesis
```

### **Step 3: Deploy the Subnet**

To deploy the Subnet locally, run:

```
avalanche subnet deploy <subnetName> -l
```

After a successful deployment, an example of what you would see:

```

Network ready to use. Local network node endpoints:
+-----+-----+
| NODE | VM | URL
+-----+-----+
| node1 | subnetName | http://127.0.0.1:9650/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc |
+-----+-----+
| node2 | subnetName | http://127.0.0.1:9652/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc |
+-----+-----+
| node3 | subnetName | http://127.0.0.1:9654/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc |
+-----+-----+
| node4 | subnetName | http://127.0.0.1:9656/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc |
+-----+-----+
| node5 | subnetName | http://127.0.0.1:9658/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc |
+-----+-----+

```

MetaMask connection details (any node URL from above works):

RPC URL: http://127.0.0.1:9650/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc

Funded address: 0x8db97C7EcE249c2b98bDC0226Cc4C2A57BF52FC with 1000000 (10^18) - private key:  
56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027

Network name: subnetName

Chain ID: 676767

Currency Symbol: SUB

Make sure to save MetaMask connection details. You will need the relevant information (RPC URL, Funded address, etc.) to interact with your Subnet.

Important thing to keep in mind is that, now that you have deployed your Subnet, it has started running in your local machine. So, after you are done interacting with your Subnet you can stop it.

To stop running the Subnet, you could run:

```
avalanche network stop
```

```
:::info
```

When you stop running the Subnet it will save the state of the Subnet and when it starts again it will continue from that state.

```
:::
```

To start running the Subnet, you could run:

```
avalanche network start
```

```
:::info
```

When you restart the Subnet RPC urls will not change. Therefore, you do not have to adjust the network in your MetaMask or anywhere else.

```
:::
```

## Step 4: Interact with the Subnet

This tutorial will cover interacting with the Subnet through [Remix](#) and [Hardhat](#).

### Step 4.1: Using Remix

First, we will be adding our Subnet to [MetaMask](#). To add the Subnet, refer to [Deploy a Smart Contract on Your Subnet-EVM Using Remix and MetaMask](#) you should replace the values with your Subnet values that are printed out after you have created it. If your balance is zero after you add Subnet to the MetaMask, refer to [Access Funded Accounts](#).

Example Values:

```
Network Name: <subnetName>
New RPC URL: http://127.0.0.1:9650/ext/bc/2KY4TYoJwuJWLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc
ChainID: 676767
Symbol: SUB
```

### Access Funded Accounts

If you followed the exact steps in this tutorial, you would see that your balance on MetaMask is zero. That is because we have only airdropped to the default account which is `0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC`. Therefore, your account on MetaMask has zero tokens and cannot send any transactions. So, to interact with the chain we have to use the address that is airdropped.

- Steps to import the airdropped account
  1. Open your MetaMask extension
  2. Click on the account image
  3. Click "Import Account"
  4. Select type: Private Key
  5. Enter private key of the default airdrop account, which is `56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027`
  6. Click "Import"

### Step 4.2: Using Hardhat

To interact with the Subnet using Hardhat, refer to [Using Hardhat with the Avalanche C-Chain](#). It is very similar to interacting with C-Chain. You only have to change `hardhat.config.ts` file. Inside that file, find the exported JavaScript object and inside of it find `networks`. Add a new network which will be your Subnet.

```
subnet: {
 url: "<yourRpcUrl>",
 chainId: <yourChainId>,
 accounts: ["<accounts-private-key>"]
}
```

Example Values:

```
subnet: {
 url: "http://127.0.0.1:9650/ext/bc/2KY4TYoJwuJWLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc",
 chainId: 676767,
 accounts: ["56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027"]
}
```

Example Updated File: [hardhat.config.ts](#)

Now you can run any commands ran in the tutorial with `--network subnet` parameter

Example command:

```
yarn deploy --network subnet
```

## Step 5: Interact with Precompiles (Optional)

If you have followed the tutorial as it is, you do not need this part. Since, in this tutorial we did not add any precompiles to the Subnet. Therefore, this step is optional and helpful only if you are trying to extend your Subnet with precompiles.

To checkout current precompiles provided by Ava Labs refer to [this](#). There are 3 precompiles shared by Ava Labs at the time this documentation is written.

- [Contract Deployer Allow List](#): restricts the addresses who can deploy contracts
- [Transaction Allow List](#): restricts the addresses who can send transactions
- [Native Minter](#): allows given addresses to mint native token

This tutorial will show how to interact with them using Remix.

:::caution

Before trying to interact with any of the precompiles make sure to add them while [Creating the Subnet](#). You can not add them afterwards.

:::

### General Steps to Interact with Precompiles

1. While creating the Subnet you will be prompted `Advanced: Would you like to add a custom precompile to modify the EVM?`: answer `Yes` then choose the precompile you would like to add. Continue by selecting `Add Admin`, it will ask for an address. This address is your account's public address, it allows others to interact with your account. To get your account's public address, open your MetaMask extension, hover over your account's label (which is 'Account 1' for the image) and click to copy the address to your clipboard. Paste that address to the command line to use your MetaMask account as the admin of the precompile. Do not forget that you can always get more details by selecting `More info` inside the command line.



:::warning

If you are adding the `Transaction allow list` precompile, make sure to add the airdrop receiver address as admin so that the address with funds could send transactions.

:::

2. Open [remix](#) and make sure that your MetaMask is using your Subnet and the remix's environment is using `Injected Web3`. Then, create a solidity file with respective recommended file name and add the respective precompile interface, refer to specific precompile to see details.
3. Load precompile to the respective address, refer to specific precompile to see their addresses.
4. Call precompile functions.

### Interact with Contract Deployer Allow List

Recommended file name: `IContractDeployerAllowList.sol`

Precompile Interface: [Contract Deployer Allow List](#)

Precompile address: `0x0200000000000000000000000000000000000000000000000000000000000000`

There are 2 main roles for Contract Deployer Allow List precompile; `Admin` and `Deployer`.

- `Admin`
  - Can add new admins and deployers
  - Can deploy contracts
- `Deployer`
  - Can deploy contracts

To check the role of an address run `readAllowList` function. It returns 0, 1 or 2, corresponding to the roles `None`, `Deployer`, and `Admin` respectively.

### Interact with Transaction Allow List

Recommended file name: `ITxAllowList.sol`

Precompile Interface: [Transaction Allow List](#)

Precompile address: `0x0200000000000000000000000000000000000000000000000000000000000002`

There are 2 main roles for Transaction Deployer Allow List precompile; `Admin` and `Allowed`.

- `Admin`
  - Can add new admins and alloweds
  - Can send transactions
- `Allowed`
  - Can send transactions

To check the role of an address run `readAllowList` function. It returns 0, 1 or 2, corresponding to the roles `None`, `Allowed`, and `Admin` respectively.

## Interact with Native Minter

Recommended file name: INativeMinter.sol

Precompile interface: [Native Minter](#)

Precompile address: 0x0200000000000000000000000000000000000000000000000000000000000000

There are 2 main roles for NativeMinter precompile; Admin and Minter .

- Admin
  - Can add new admins and minters
  - Can mint native token
- Minter
  - Can mint native token

To check the role of an address run `readAllowList` function. It returns 0, 1 or 2, corresponding to the roles `None`, `Minter`, and `Admin` respectively.

## Customize the Subnet

- VM : To understand and create your custom VM you can refer to [this](#).
- ChainId : You want your `ChainId` parameter to be unique. To make sure that your Subnet is secure against replay attacks. To see registered ChainIds you can check [chainlist.org](#). At the top right of the site make sure to turn on the button to include testnets.
- Gas Parameters : Ava Labs recommends the low-low option and C-Chain currently uses this option. But, if you know what you are doing you are free to customize. Note that higher disk usage has some trade offs, it would require more processing power and cause it to be more expensive to maintain.
- Airdrop Address : You would not like to use the default address in production, that is receiving the 1 million tokens. Because, it is a compromised wallet, which means that its private key is well known by others. If you add a custom address to receive airdrop. Avalanche-cli will ask you to give an amount in AVAX, in that case do not enter the value thinking as in `ether` but in `gwei` to correctly airdrop the amount you want. As an example, to airdrop 1 whole token, as in one ether, you would enter the value `1000000000`.
- Precompiles : You can learn what precompiles are by referring to [this](#).

## Troubleshoot Common Issues

### Step 1: Install Avalanche-Cli

- `avalanche` , `avalanche subnet`
  - "command not found: avalanche" It means that the directory containing `avalanche` command is not added to the PATH environment variable. It could be caused by following reasons;
    - You have added the wrong `bin` directory to your environment variables.
      - Make sure to find where the `bin` directory is and run `export PATH=$PWD:$PATH` inside the `bin` directory.
    - You have added it to the PATH environment variable temporarily and restarted your terminal.
      - You can either add the `bin` directory to the PATH environment variable again by running `export PATH=$PWD:$PATH` inside the downloaded `bin` directory or you can refer to [Add Avalanche Command Permanently](#)

### Step 2: Create the Subnet

- `avalanche subnet create <subnetName>`
  - "Error: configuration already exists." It means that you have already created a Subnet with the same name. To check if that is the case, you can run `avalanche subnet list` which would list the Subnets you have. If you have a Subnet with the same name, you can try to create with a different name, delete the existing Subnet by running `avalanche subnet delete <subnetName>` or overwrite the existing one by running `avalanche subnet create <subnetName> --force`

### Step 3: Deploy the Subnet

- `avalanche subnet deploy <subnetName> -l`
  - "Subnet <subnetName> has already been deployed" As it says, it means that your Subnet has already been deployed. To check currently running blockchains run `avalanche network status` . If it provides network information, try to connect to your Subnet using MetaMask to check if everything's all right. If it does not provide network information or you are having problems with interacting with your Subnet. Run `avalanche network clean` , this command will stop the local network and delete the state. Then, run `avalanche subnet deploy <subnetName> -l` again.
  - "Error: failed to query network health: ..." You can check logs which are located at `$HOME/.avalanche-cli` or try to run the command once more.

### Step 4: Interact with the Subnet

- Errors are generally covered in the linked tutorials. Be sure to check them out.
- When you try to interact with the Subnet you might try to interact with an account that has no balance. Make sure that you have followed [Access Funded Accounts](#). If you are having a problem interacting using Hardhat, make sure that the private key corresponds to an account which has balance.

### Step 5: Interact with Precompiles (Optional)

- Common issues are troubleshooted at the official avalanche docs, to check them out refer to [this](#).
  - For Contract Deployer Allow List , refer to [this](#).
  - For Transaction Allow List , refer to [this](#).

- o For Native Minter , refer to [this](#).

## Conclusion

That is it! That is how you could create and deploy your local Subnet from scratch.

In this tutorial, we learned:

- Installing and using avalanche-cli.
- Creating a Subnet and customizing it.
- Deploying the Subnet locally for local development.
- Interacting with locally deployed Subnet using Remix and Hardhat.
- Optionally, we learned how to interact with precompiles.

## hardhat.config.js

```
import { task } from "hardhat/config";
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
import { BigNumber } from "ethers";
import "@nomiclabs/hardhat-waffle";

// When using the hardhat network, you may choose to fork Fuji or Avalanche Mainnet
// This will allow you to debug contracts using the hardhat network while keeping the current network state
// To enable forking, turn one of these booleans on, and then run your tasks/scripts using `--network hardhat` -
// For more information go to the hardhat guide
// https://hardhat.org/hardhat-network/
// https://hardhat.org/guides/mainnet-forking.html
const FORK_FUJI = false;
const FORK_MAINNET = false;
const forkingData = FORK_FUJI
 ?
 {
 url: "https://api.avax-test.network/ext/bc/C/rpc",
 }
 : FORK_MAINNET
 ?
 {
 url: "https://api.avax.network/ext/bc/C/rpc",
 }
 : undefined;

task(
 "accounts",
 "Prints the list of accounts",
 async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners();
 accounts.forEach((account: SignerWithAddress): void => {
 console.log(account.address);
 });
 }
);

task(
 "balances",
 "Prints the list of AVAX account balances",
 async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners();
 for (const account of accounts) {
 const balance: BigNumber = await hre.ethers.provider.getBalance(
 account.address
);
 console.log(`${account.address} has balance ${balance.toString()}`);
 }
 }
);

export default {
 solidity: {
 compilers: [
 {
 version: "0.5.16",
 },
 {
 version: "0.6.2",
 },
 {
 version: "0.6.4",
 },
],
 },
}
```

```

 version: "0.7.0",
 },
 {
 version: "0.8.0",
 },
],
},
networks: {
 hardhat: {
 gasPrice: 225000000000,
 chainId: !forkingData ? 43112 : undefined, //Only specify a chainId if we are not forking
 forking: forkingData,
 },
 local: {
 url: "http://localhost:9650/ext/bc/C/rpc",
 gasPrice: 225000000000,
 chainId: 43112,
 accounts: [
 "0x56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027",
 "0x7b4198529994ab0dc604278c99d153cf069d594753d471171a1d102a10438e07",
 "0x15614556be13730e9e8d6eacc1603143e7b96987429df8726384c2ec4502ef6e",
 "0x31b571bf6894a248831ff937bb49f7754509fe93bbd2517c9c73c4144c0e97dc",
 "0x6934bef917e01692b789da754a0eae31a8536eb465e7bff752ea291dad88c675",
 "0xe700bdbdbc79b808b1ec45f8c2370e4616d3a02c336e68d85d4668e08f53cff",
 "0xb0c2865b76ba28016bc2255c7504d000e046ae01934b04c694592a6276988630",
 "0xcdbfd34f687ced8c6968854f8a99ae47712cd4f4183b78dcc44903dlbfe8cbf60",
 "0x86f78c5416151fe3546dece84fda4b4b1e36089f2dbc48496faf3a950f16157c",
 "0x750839e9dbbd2a0910efe40f50b2f3b2f2f59f5580bb4b83bd8c1201cf9a010a",
],
 },
 fuji: {
 url: "https://api.avax-test.network/ext/bc/C/rpc",
 gasPrice: 225000000000,
 chainId: 43113,
 accounts: [],
 },
 mainnet: {
 url: "https://api.avax.network/ext/bc/C/rpc",
 gasPrice: 225000000000,
 chainId: 43114,
 accounts: [],
 },
},
// ONLY CHANGED PART STARTS
subnet: {
 url: "http://127.0.0.1:9650/ext/bc/2KY4TYoJwuJWLLeSfj4Mae4t4sBCwGrx48QyGWg3zWwP5PhZjHV/rpc",
 chainId: 676767,
 accounts: [
 "56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027",
],
},
// ONLY CHANGED PART ENDS
},
);

```

## AllowList

```

// (c) 2022-2023, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

interface AllowListInterface {
 // Set [addr] to have the admin role over the allow list
 function setAdmin(address addr) external;

 // Set [addr] to be enabled on the allow list
 function setEnabled(address addr) external;

 // Set [addr] to have no role over the allow list
 function setNone(address addr) external;

 // Read the status of [addr]
}
```

```

function readAllowList(address addr) external view returns (uint256);
}

```

## NativeMinter

```

// (c) 2022-2023, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

interface NativeMinterInterface {
 // Set [addr] to have the admin role over the minter list
 function setAdmin(address addr) external;

 // Set [addr] to be enabled on the minter list
 function setEnabled(address addr) external;

 // Set [addr] to have no role over the minter list
 function setNone(address addr) external;

 // Read the status of [addr]
 function readAllowList(address addr) external view returns (uint256);

 // Mint [amount] number of native coins and send to [addr]
 function mintNativeCoin(address addr, uint256 amount) external;
}

```

## Contest Overview

| Title                                  | Description                                        |
|----------------------------------------|----------------------------------------------------|
| <a href="#">2022 Tutorial Contests</a> | List of winning tutorials in 2022 tutorial contest |
| <a href="#">2021 Tutorial Contests</a> | List of winning tutorials in 2021 tutorial contest |

:::warning

These tutorials are published as a snapshot when they were written. For up-to-date information, please reach out to the owners of these projects.

:::

## Developer Toolchains Overview

| Title                                                                 | Description                                                                                   |
|-----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <a href="#">Using Foundry with the Avalanche C-Chain</a>              | Using Foundry with the Avalanche's C-Chain, an instance of the Ethereum Virtual Machine (EVM) |
| <a href="#">Using Hardhat with the Avalanche C-Chain</a>              | Using Hardhat with the Avalanche's C-Chain, an instance of the Ethereum Virtual Machine (EVM) |
| <a href="#">Using Truffle with the Avalanche C-Chain</a>              | Using Truffle with the Avalanche's C-Chain, an instance of the Ethereum Virtual Machine (EVM) |
| <a href="#">Verifying Smart Contracts Using Hardhat and Snowtrace</a> | Verifying Smart Contracts Using Hardhat and Snowtrace                                         |
| <a href="#">Verifying Smart Contracts with Truffle Verify</a>         | Verify Smart Contracts with Sourcify and Truffle                                              |

## Using Foundry with the Avalanche C-Chain

### Introduction

This guide shows how to deploy and interact with smart contracts using foundry on a local Avalanche Network and the [Fuji C-Chain](#), which is an instance of the EVM.

[Foundry toolchain](#) is a smart contract development toolchain written in Rust. It manages your dependencies, compiles your project, runs tests, deploys, and lets you interact with the chain from the command-line.

### Recommended Knowledge

- Basic understanding of [Solidity](#) and Avalanche.

- You are familiar with [Avalanche Smart Contract Quickstart](#).
- Basic understanding of the [Avalanche's architecture](#)
- performed a cross-chain swap via this [this tutorial](#) to get funds to your C-Chain address.

## Requirements

- You have [installed Foundry](#) and run `foundryup`. This installation includes the `forge` and `cast` binaries used in this walk-through.
- [NodeJS](#) version 16.x

### AvalancheGo and Avalanche Network Runner

[AvalancheGo](#) is an Avalanche node implementation written in Go.

[Avalanche Network Runner](#) is a tool to quickly deploy local test networks. Together, you can deploy local test networks and run tests on them.

Start a local five node Avalanche network:

```
cd /path/to/avalanche-network-runner
start a five node staking network
./go run examples/local/fivenodenetwork/main.go
```

A five node Avalanche network is running on your machine. Network will run until you Ctrl + C to exit.

## Getting Started

This section will walk you through creating an [ERC721](#).

### Clone Avalanche Smart Contract Quick Start

Clone the [quickstart repository](#) and install the necessary packages via `yarn`.

```
git clone https://github.com/ava-labs/avalanche-smart-contract-quickstart.git
cd avalanche-smart-contract-quickstart
yarn
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanche-smart-contract-quickstart.git
```

You can find more about SSH and how to use it [here](#). :::

In order to deploy contracts, you need to have some AVAX. You can get testnet AVAX from the [Avalanche Faucet](#), which is an easy way to get to play around with Avalanche. After getting comfortable with your code, you can run it on Mainnet after making the necessary changes to your workflow.

## Write Contracts

We will use our example ERC721 smart contract, [NFT.sol](#), found in `./contracts` of our project.

```
// SPDX-License-Identifier: MIT
// contracts/ERC721.sol

pragma solidity >=0.6.2;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract NFT is ERC721 {
 using Counters for Counters.Counter;
 Counters.Counter private _tokenIds;

 constructor() ERC721("GameItem", "ITM") {}

 // commented out unused variable
 // function awardItem(address player, string memory tokenURI)
 function awardItem(address player)
 public
 returns (uint256)
 {
 _tokenIds.increment();

 uint256 newItemId = _tokenIds.current();
 _mint(player, newItemId);
 // _setTokenURI(newItemId, tokenURI);

 return newItemId;
 }
}
```

Let's examine this implementation of an NFT as a Game Item. We start by importing to contracts from our node modules. We import OpenZeppelin's open source implementation of the [ERC721 standard](#) which our NFT contract will inherit from. Our constructor takes the `_name` and `_symbol` arguments for our NFT and passes them on to the constructor of the parent ERC721 implementation. Lastly we implement the `awardItem` function which allows anyone to mint an NFT to a player's wallet address. This function increments the `currentTokenId` and makes use of the `_mint` function of our parent contract.

## Compile, Deploy, and Verify with Forge

[Forge](#) is a command-line tool that ships with Foundry. Forge tests, builds, and deploys your smart contracts.

It requires some initial project configuration in the form of a [foundry.toml](#) which can be generated by running:

```
forge init --no-git --no-commit --force
```

The `foundry.toml` by default points to the folders it added. We will want to change this to make sure the `src` points to the `contracts` directory. Change your `foundry.toml` to look like the following:

```
[profile.default]
src = 'contracts'
out = 'out'
libs = ["node_modules", "lib"]
remappings = [
 '@ensdomains/=node_modules/@ensdomains/',
 '@openzeppelin/=node_modules/@openzeppelin/',
 'hardhat/=node_modules/hardhat/',
]

See more config options https://github.com/foundry-rs/foundry/tree/master/config
```

To compile the NFT contract run:

```
forge build
```

By default, the contract artifacts will be in the `out` directory, as specified in the `foundry.toml`. To deploy our compiled contract with Forge we have to set environment variables for the RPC endpoint and the private key we want to use to deploy.

Set your environment variables by running:

```
export RPC_URL=<YOUR-RPC-ENDPOINT>
export PRIVATE_KEY=<YOUR-PRIVATE-KEY>
```

Since we are deploying to Fuji testnet, our `RPC_URL` export should be:

```
export RPC_URL=https://api.avax-test.network/ext/bc/C/rpc
```

If you would like to verify your contracts during the deployment process (fastest and easiest way), get a [Snowtrace API Key](#). Add this as an environment variable:

```
export ETHERSCAN_API_KEY=<YOUR-SNOWTRACE-API-KEY>
```

Once set, you can [deploy your NFT with Forge](#) by running the command below while adding the values for `_name` and `_symbol`, the relevant [constructor arguments](#) of the NFT contract. You can verify the contracts with Snowtrace by adding `--verify` before the `--constructor-args`:

```
forge create NFT --rpc-url=$RPC_URL --private-key=$PRIVATE_KEY --verify --constructor-args GameItem ITM
```

Upon successful deployment, you will see the deploying wallet's address, the contract's address as well as the transaction hash printed to your terminal.

Here's an example output from an NFT deployment and verification.

```
[#] Compiling...
No files changed, compilation skipped
Deployer: 0x8db97c7cece249c2b98bcd0226cc4c2a57bf52fc
Deployed to: 0x52c84043cd9c865236f11d9fc9f56aa003c1f922
Transaction hash: 0xf35c40dbbd9e4298698ad1cb9937195e5a5e74e557bab1970a5dfd42a32f533
```

Upon successful verification, after your deployment you will see the contract verification status as `successfully verified`:

```
Starting contract verification...
Waiting for etherscan to detect contract deployment...
Start verifying contract `0x8e982a4ef70430f8317b5652bd5c28f147fb912` deployed on fuji

Submitting verification for [contracts/NFT.sol:NFT] "0x8e982a4Ef70430f8317B5652Bd5C28F147FBf912".

Submitting verification for [contracts/NFT.sol:NFT] "0x8e982a4Ef70430f8317B5652Bd5C28F147FBf912".
Submitted contract for verification:
```

```
Response: `OK`
GUID: `cfkyqwjauafirepxt8qhks2zhtptzzccqgege9uefu9ma8wiz`
URL:
 https://testnet.snowtrace.io/address/0x8e982a4ef70430f8317b5652bd5c28f147fbf912

Contract verification status:
Response: `NOTOK`
Details: `Pending in queue`

Contract verification status:
Response: `OK`
Details: `Pass - Verified`

Contract successfully verified
```

Note: Please store your Deployed to address for use in the next sections.

## Verifying After Deployment

If you did not verify within the deployment process, you can still verify a deployed contract with foundry, using `forge verify-contract`.

*Note: The foundry.toml and environment variables will have to be set like they were in the previous section.*

For example, if we were to verify the NFT contract we just deployed in the previous section it would look this:

```
forge verify-contract --chain-id 43113 --watch --constructor-args $(cast abi-encode "constructor(string,string)" "GameItem" "ITM") 0x8e982a4ef70430f8317b5652bd5c28f147fb912 NFT
```

Upon successful verification, you will see the contract verification status as successfully verified.

```
Starting contract verification...
Waiting for etherscan to detect contract deployment...
Start verifying contract `0x8e982a4ef70430f8317b5652bd5c28f147fbf912` deployed on fuji

Submitting verification for [contracts/NFT.sol:NFT] "0x8e982a4Ef70430f8317B5652Bd5C28F147FBf912".

Submitting verification for [contracts/NFT.sol:NFT] "0x8e982a4Ef70430f8317B5652Bd5C28F147FBf912".
Submitted contract for verification:
 Response: `OK`
 GUID: `cfkyqwjauafirepxt8qhks2zhptczccqge9uefu9ma8wiz`
 URL:
 https://testnet.snowtrace.io/address/0x8e982a4ef70430f8317b5652bd5c28f147fbf912
Contract verification status:
Response: `NOTOK`
Details: `Pending in queue`
Contract verification status:
Response: `OK`
Details: `Pass - Verified`
Contract successfully verified
```

## Using Cast to Interact with the Smart Contract

We can call functions on our NFT contract with [Cast](#), Foundry's command-line tool for interacting with smart contracts, sending transactions, and getting chain data. In this scenario, we will mint a Game Item to a player's wallet using the [awardItem](#) function in our smart contract.

Mint an NFT from your contract by replacing <NFT-CONTRACT-ADDRESS> with your Deployed to address and <NFT-RECIPIENT-ADDRESS> with an address of your choice.

*Note: This section assumes that you have already set your RPC and private key env variables during deployment.*

```
cast send --rpc-url=$RPC_URL <NFT-CONTRACT-ADDRESS> "awardItem(address)" <NFT-RECIPIENT-ADDRESS> --private-key=$PRIVATE_KEY
```

Upon success, the command line will display the transaction data.

|                  |   |
|------------------|---|
| transactionIndex | 0 |
| type             | 2 |

Well done! You just minted your first NFT from your contract. You can check the owner of tokenId 1 by running the `cast call` command below:

```
cast call --rpc-url=$RPC_URL --private-key=$PRIVATE_KEY <NFT-CONTRACT-ADDRESS> "ownerOf(uint256)" 1
```

The address you provided above should be returned as the owner.

## Mainnet Workflow

The Fuji workflow above can be adapted to Mainnet with the following modifications to the environment variables:

```
export RPC_URL=https://api.avax.network/ext/bc/C/rpc
export PRIVATE_KEY=<YOUR-PRIVATE-KEY>
```

## Local Workflow

The Fujii workflow above can be adapted to a Local Network by doing following:

In a new terminal navigate to your [Avalanche Network Runner](#) directory.

```
cd /path/to/Avalanche-Network-Runner
```

Next, deploy a new Avalanche Network with five nodes (a Cluster) locally.

```
go run examples/local/fivenodenetwork/main.go
```

Next, modify the environment variables in your Foundry project:

```
export RPC_URL=http://localhost:9650/ext/bc/C/rpc
export PRIVATE_KEY=56289e99c94b6912bfcc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027
```

:::warning

The example PRIVATE\_KEY variable above provides a pre-funded account on Avalanche Network Runner and should be used for LOCAL DEVELOPMENT ONLY.

•

## Summary

Now you have the tools you need to launch a local Avalanche network, create a Foundry project, as well as create, compile, deploy and interact with Solidity contracts.

Join our [Discord Server](#) to learn more and ask any questions you may have.

## Using Hardhat with the Avalanche C-Chain

## Introduction

Avalanche is an open-source platform for launching decentralized applications and enterprise blockchain deployments in one interoperable, highly scalable ecosystem. Avalanche gives you complete control on both the network and application layers—helping you build anything you can imagine.

The Avalanche Network is composed of many blockchains. One of these blockchains is the C-Chain (Contract Chain), which is an Ethereum Virtual Machine instance. The C-Chain's API is almost identical to an Ethereum node's API. Avalanche offers the same interface as Ethereum but with higher speed, higher throughput, lower fees and lower transaction confirmation times. These properties considerably improve the performance of DApps and the user experience of smart contracts.

The goal of this guide is to lay out best practices regarding writing, testing and deployment of smart contracts to Avalanche's C-Chain. We'll be building smart contracts with development environment [Hardhat](#).

## Prerequisites

Node.js and Yarn

First, install the LTS (long-term support) version of [NodeJS](#). This is 18.x at the time of writing. NodeJS bundles npm

Next, install [yarn](#):

```
npm install -g yarn
```

## AvalancheGo and Avalanche Network Runner

[AvalancheGo](#) is an Avalanche node implementation written in Go. [Avalanche Network Runner](#) is a tool to quickly deploy local test networks. Together, you can deploy local test networks and run tests on them.

## Solidity and Avalanche

It is also helpful to have a basic understanding of [Solidity](#) and Avalanche.

## Dependencies

Clone the [quickstart repository](#) and install the necessary packages via `yarn`.

```
git clone https://github.com/ava-labs/avalanche-smart-contract-quickstart.git
cd avalanche-smart-contract-quickstart
yarn
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanche-smart-contract-quickstart.git
```

You can find more about SSH and how to use it [here](#). :::

## Write Contracts

Edit the `ExampleERC20.sol` contract in `contracts/`. `ExampleERC20.sol` is an [Open Zeppelin ERC20](#) contract. ERC20 is a popular smart contract interface. You can also add your own contracts.

## Hardhat Config

Hardhat uses `hardhat.config.js` as the configuration file. You can define tasks, networks, compilers and more in that file. For more information see [here](#).

Here is an example pre-configured `hardhat.config.ts`.

```
import { task } from "hardhat/config"
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers"
import { BigNumber } from "ethers"
import "@nomiclabs/hardhat-waffle"

// When using the hardhat network, you may choose to fork Fuji or Avalanche Mainnet
// This will allow you to debug contracts using the hardhat network while keeping the current network state
// To enable forking, turn one of these booleans on, and then run your tasks/scripts using `--network hardhat`-
// For more information go to the hardhat guide
// https://hardhat.org/hardhat-network/
// https://hardhat.org/guides/mainnet-forking.html
const FORK_FUJI = false
const FORK_MAINNET = false
const forkingData = FORK_FUJI ? {
 url: 'https://api.avax-test.network/ext/bc/C/rpc',
} : FORK_MAINNET ? {
 url: 'https://api.avax.network/ext/bc/C/rpc'
} : undefined

export default {
 solidity: {
 compilers: [
 {
 version: "0.5.16"
 },
 {
 version: "0.6.2"
 },
 {
 version: "0.6.4"
 },
 {
 version: "0.7.0"
 },
 {
 version: "0.8.0"
 }
]
 },
 networks: {
 hardhat: {
 gasPrice: 225000000000,
 chainId: !forkingData ? 43112 : undefined, //Only specify a chainId if we are not forking
 forking: forkingData
 }
 }
}
```

```

},
local: {
 url: 'http://localhost:9650/ext/bc/C/rpc',
 gasPrice: 225000000000,
 chainId: 43112,
 accounts: [
 "0x56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027",
 "0xb4198529994b0dc604278c99d153cf069d594753d471171a1d102a10438e07",
 "0x15614556be13730e9e8d6eacc1603143e7b96987429df8726384c2ec4502ef6e",
 "0x31b571bf6894a248831ff937bb49f754509fe93bbd2517c9c73c4144c0e97dc",
 "0x6934bef917e01692b789da754a0eae31a8536eb465e7bff752ea291dad88c675",
 "0xe700bdbdb279b808b1ec45f8c2370e4616d3a02c336e68d85d4668e08f53cff",
 "0xbcb2865b76ba28016bc225c7504d00e046ae01934b04c694592a6276988630",
 "0xcd8fd34f687ced8c6968854f8a99ae47712c4f4183b78dcc4a903d1bfe8cbf60",
 "0x86f78c5416151fe3546dece84fd4b4b1e36089f2dbc48496faf3a950f16157c",
 "0x750839e9dbbd2a0910efe40f50b2f3b2f59f5580bb4b83bd8c1201cf9a010a"
]
},
fuji: {
 url: 'https://api.avax-test.network/ext/bc/C/rpc',
 gasPrice: 225000000000,
 chainId: 43113,
 accounts: []
},
mainnet: {
 url: 'https://api.avax.network/ext/bc/C/rpc',
 gasPrice: 225000000000,
 chainId: 43114,
 accounts: []
}
}
}

```

This configures necessary network information to provide smooth interaction with Avalanche. There are also some pre-defined private keys for testing on a local test network.

:::info

The port in this tutorial uses 9650. Depending on how you start your local network, it could be different.

:::

## Hardhat Tasks

You can define custom hardhat tasks in `hardhat.config.ts`. There are two tasks included as examples: `accounts` and `balances`.

```

task("accounts", "Prints the list of accounts", async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners()
 accounts.forEach((account: SignerWithAddress): void => {
 console.log(account.address)
 })
}

task("balances", "Prints the list of AVAX account balances", async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners()
 for(const account of accounts){
 const balance: BigNumber = await hre.ethers.provider.getBalance(
 account.address
);
 console.log(` ${account.address} has balance ${balance.toString()}`);
 }
})

```

`npx hardhat accounts` prints the list of accounts. `npx hardhat balances` prints the list of AVAX account balances. As with other `yarn` scripts you can pass in a `--network` flag to hardhat tasks.

## Accounts

Prints a list of accounts on the local Avalanche Network Runner network.

```

npx hardhat accounts --network local
0x8db97C7EcE249c2b98bdC0226Cc4C2A57BF52FC
0x9632a7965af553F58738B0FB750320158495942
0x55ee05dF718f1a5C1441e76190EB1a19e2C9430
0x4cf2eD3665F6bFA95cE6A11CFDb7A2EF5FC1C7E4
0x0B891db1901d4875056896f28B666508393C7A8
0x01F253bE2EBF0bd64649FA468bF7b95ca933BDe2
0x78A23300E04FB5d2820E23cc679738982e1fd5

```

```
0x3C7daE394BBf8e9EE1359ad14C1c47003bD06293
0x61e0B3CD93F36847Abbd5d40d6F00a8ec6f3cffB
0x0Fa8EA536Be85F32724D57A37758761B86416123
```

## Balances

Prints a list of accounts and their corresponding AVAX balances on the local Avalanche Network Runner network.

```
npx hardhat balances --network local
0x8db97C7EcE249c2b98DC0226Cc4C2A57BF52FC has balance 500
0x9632a79656af553F58738B0FB750320158495942 has balance 0
0x55ee005df718f1a5c1441e76190EB1a19e2C9430 has balance 0
0x4cf2ed3661F6bFA95cE6A11CFDb7A2EF5FC1C7E4 has balance 0
0x0B891dB1901D4875056896f28B6665083935C7A8 has balance 0
0x01F253bE2EBF0bd64649FA468bFTb95ca933BDe2 has balance 0
0x78A23300E04FB5d5D2820E23cc679738982e1fd5 has balance 0
0x3C7daE394BBf8e9EE1359ad14C1c47003bD06293 has balance 0
0x61e0B3CD93F36847Abbd5d40d6F00a8ec6f3cffB has balance 0
0x0Fa8EA536Be85F32724D57A37758761B86416123 has balance 0
```

Notice that the first account is already funded. This is because this address is pre-funded in the local network genesis file.

## ERC20 Balances

```
task("check-erc20-balance", "Prints out the ERC20 balance of your account").setAction(async function (taskArguments, hre) {
 const genericErc20Abi = require("./erc20.abi.json");
 const tokenContractAddress = "0x...";
 const provider = ethers.getDefaultProvider("https://api.avax.network/ext/bc/C/rpc");
 const contract = new ethers.Contract(tokenContractAddress, genericErc20Abi, provider);
 const balance = await contract.balanceOf("0x...");
 console.log(`Balance in wei: ${balance}`)
});
```

This will return the result in wei. If you want to know the exact amount of token with its token name then you need to divide it with its decimal. `erc20.abi.json` can be [found here](#).

The example uses the [C-Chain Public API](#) for the provider. For a local Avalanche network use `http://127.0.0.1:9650/ext/bc/C/rpc` and for Fuji Testnet use `https://api.avax-test.network/ext/bc/C/rpc`.

## Hardhat Help

Run `yarn hardhat` to list Hardhat's version, usage instructions, global options and available tasks.

## Typical Avalanche Network Runner Workflow

### Run Avalanche Network Runner

First confirm you have the latest AvalancheGo built.

```
cd /path/to/avalanchego
git fetch -p
git checkout master
../scripts/build.sh
```

(Note that you can also [download pre-compiled AvalancheGo binaries](#) rather than building from source.)

Confirm you have Avalanche Network Runner installed by following the steps listed [here](#)

Start Avalanche Network Runner and run a script to start a new local network.

### Start the Server

```
$ cd /path/to/Avalanche-Network-Runner
$ avalanche-network-runner server \
--log-level debug \
--port=:8080" \
--grpc-gateway-port=:8081"
```

### Start a New Avalanche Network with Five Nodes

```
replace execPath with the path to AvalancheGo on your machine
e.g., ${HOME}/go/src/github.com/ava-labs/avalanchego/build/avalanchego
$ AVALANCHEGO_EXEC_PATH="avalanchego"
```

```
$ avalanche-network-runner control start \
--log-level debug \
--endpoint="0.0.0.0:8080" \
--number-of-nodes=5 \
--avalanche-go-path ${AVALANCHEGO_EXEC_PATH}
```

Now you're running a local Avalanche network with 5 nodes.

### Fund Accounts

Transfer 1,000 AVAX from the X-Chain to each of the 10 accounts in `hardhat.config.ts` with the script [fund-cchain-addresses](#). Funding these accounts is a prerequisite for deploying and interacting with smart contracts.

Note: If you see `Error: Invalid JSON RPC response: "API call rejected because chain is not done bootstrapping"`, you need to wait until network is bootstrapped and ready to use. It should not take too long.

```
$ cd /path/to/avalanche-smart-contract-quickstart
$ yarn fund-cchain-addresses
yarn run v1.22.4
npx hardhat run scripts/fund-cchain-addresses.js
Exporting 1000 AVAX to each address on the C-Chain...
2b75ae74ScLkWe5GVFTYJoP2EniMywkcZySQuoFGN2EJLiPDgp
Importing AVAX to the C-Chain...
2dyXcQGiC1ckCX4fs8LgL8GJgsM72f9Ga13rX5v9TAgvVJYm
✨ Done in 5.03s.
```

Confirm each of the accounts are funded with 1000 AVAX.

```
$ yarn balances --network local
yarn run v1.22.4
npx hardhat balances --network local
0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC has balance 50000001000000000000000000000000
0x9632a79656af553F58738B0FB750320158495942 has balance 10000000000000000000000000000000
0x55ee05d718f1a5C1441e76190EB1a19eE2C9430 has balance 10000000000000000000000000000000
0x4cf2e3665F6bFA95cE6A11CFDb7A2EF5FC1C7E4 has balance 10000000000000000000000000000000
0x0B891d1B901D4875056896f28B6665083935C7A8 has balance 10000000000000000000000000000000
0x01F253bE2EBF0bd64649FA468bF7b95ca933BDc2 has balance 10000000000000000000000000000000
0x78A23300E04FB5d5D2820E23cc679738982e1fd5 has balance 10000000000000000000000000000000
0x3C7daE394BBf8e9EE1359ad14C1c47003bD06293 has balance 10000000000000000000000000000000
0x61e0B3CD93F36847Abbd5d40d6F00a8c6f3cfFB has balance 10000000000000000000000000000000
0x0Fa8EA536Be85F32724D57A37758761B86416123 has balance 10000000000000000000000000000000
✨ Done in 0.72s.
```

Send each of the accounts some AVAX from the first account.

```
$ yarn send-avax-wallet-signer --network local
yarn run v1.22.4
npx hardhat run scripts/sendAvaWalletSigner.ts --network local
Seeding addresses with AVAX
✨ Done in 1.33s.
```

Confirm that the balances are updated

```
$ yarn balances --network local
yarn run v1.22.4
npx hardhat balances --network local
0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC has balance 4999999999527500000000000000
0x9632a79656af553F58738B0FB750320158495942 has balance 10000100000000000000000000000000
0x55ee05d718f1a5C1441e76190EB1a19eE2C9430 has balance 10000100000000000000000000000000
0x4cf2e3665F6bFA95cE6A11CFDb7A2EF5FC1C7E4 has balance 10000100000000000000000000000000
0x0B891d1B901D4875056896f28B6665083935C7A8 has balance 10000100000000000000000000000000
0x01F253bE2EBF0bd64649FA468bF7b95ca933BDc2 has balance 10000100000000000000000000000000
0x78A23300E04FB5d5D2820E23cc679738982e1fd5 has balance 10000100000000000000000000000000
0x3C7daE394BBf8e9EE1359ad14C1c47003bD06293 has balance 10000100000000000000000000000000
0x61e0B3CD93F36847Abbd5d40d6F00a8c6f3cfFB has balance 10000100000000000000000000000000
0x0Fa8EA536Be85F32724D57A37758761B86416123 has balance 10000100000000000000000000000000
```

Note: If you see `Error HH108: Cannot connect to the network local. Please make sure your node is running, and check your internet connection and networks config , ensure that you are using a valid Node Port. See which ports the Nodes are using by running the command:`

```
$ cd /path/to/avalanche-network-runner
$ avalanche-network-runner control uris \
--log-level debug \
--endpoint="0.0.0.0:8080"
```

## Compile Smart Contracts

In [package.json](#) there's a `compile` script.

```
"compile": "npx hardhat compile",
```

Run `yarn compile` to make sure your project compiles.

Compile the smart contract.

```
$ yarn compile
yarn run v1.22.4
rimraf ./build/
npx hardhat compile
Compiling 1 file with 0.6.4
Compilation finished successfully
✨ Done in 2.13s.
```

## Deploy Smart Contracts

Hardhat enables deploying to multiple environments. In [package.json](#) there is a script for deploying.

Edit the deployment script in `scripts/deploy.ts`

```
"deploy": "npx hardhat run scripts/deploy.ts",
```

You can choose which environment that you want to deploy to by passing in the `--network` flag with `local` (for example a local network created with Avalanche Network Runner), `fuji`, or `mainnet` for each respective environment. If you don't pass in `--network` then it will default to the hardhat network. For example, if you want to deploy to Mainnet:

```
yarn deploy --network mainnet
```

Deploy the contract to your local network

```
$ yarn deploy --network local
yarn run v1.22.4
npx hardhat run scripts/deploy.ts --network local
Coin deployed to: 0x17aB05351fC94a1a67Bf3f56DdbB941aE6
✨ Done in 1.28s.
```

We now have a token deployed at `0x17aB05351fC94a1a67Bf3f56DdbB941aE6`.

## Interact with Smart Contract

Hardhat has a developer console to interact with contracts and the network. For more information about Hardhat's console see [here](#). Hardhat console is a NodeJS-REPL, and you can use different tools in it. [Ethers](#) is the library that we'll use to interact with our network.

You can open console with:

```
$ yarn console --network local
yarn run v1.22.11
npx hardhat console --network local
Welcome to Node.js v16.2.0.
Type ".help" for more information.
>
```

Get the contract instance with factory and contract address to interact with our contract:

```
> const Coin = await ethers.getContractFactory('ExampleERC20');
undefined
> const coin = await Coin.attach('0x17aB05351fC94a1a67Bf3f56DdbB941aE6')
undefined
```

The first line retrieves contract factory with ABI & bytecode. The second line retrieves an instance of that contract factory with given contract address. Recall that our contract was already deployed to `0x17aB05351fC94a1a67Bf3f56DdbB941aE6` in the previous step.

Fetch the accounts:

```
> let accounts = await ethers.provider.listAccounts()
undefined
> accounts
[
 '0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC',
 '0x9632a79656af553F58738B0FB750320158495942',
```

```
'0x55ee05fD7F18f1a5C1441e76190EB1a19eE2C9430',
'0x4cf2eD365f6FB9A95cE6A11CFD7a2E5F2C17E4',
'0x08B91d9101D4785065896f2B8665083935C7A8',
'0x01F253B2eE8FB0d64649FA468fb7b95ca393Bd2E',
'0x78A23300E04Bf5d5B282E02c3c679738982e1fd5',
'0x3C7daE394BBF8e9EE1359ad14C1C47003bD06293',
'0x6le0B3CD93F36847Abbd54d0d6F00a8eC6f3cffB',
'0x0Fa8EA536Be8F532724D57A37758761B86416123'
]
```

This is exactly the same account list as in yarn accounts .

Now we can interact with our ERC-20 contract:

```
> let value = await coin.balanceOf(accounts[0])
undefined
> value.toString()
'123456789'
> value = await coin.balanceOf(accounts[1])
BigNumber { _hex: '0x00', _isBigNumber: true }
> value.toString()
'0'
```

account[0] has a balance because account[0] is the default account. The contract is deployed with this account. The constructor of [ERC20.sol](#) mints TOTAL\_SUPPLY of 123456789 token to the deployer of the contract.

`accounts[1]` currently has no balance. Send some tokens to `accounts[1]`, which is `0x9632a79656af553F58738B0FB750320158495942`.

Note: Since this is a local network, we did not need to wait until transaction is accepted. However for other networks like `fuji` or `mainnet` you need to wait until transaction is accepted with `await tx.wait()`.

No one can accuse that station as a transformer.

```
> value = await coin.balanceOf(accounts[0])
BigNumber { _hex: '0x075bccb1', _isBigNumber: true }
> value.toString()
'123456689'
> value = await coin.balanceOf(accounts[1])
BigNumber { _hex: '0x64', _isBigNumber: true }
> value.toString()
'100'
```

As you might have noticed there was no "sender" information in `await coin.transfer(accounts[1], 100)`; this is because `ethers` uses the first signer as the default signer. In our case this is `accounts[0]`. If we want to use another account we need to connect with it first.

```
> let signer1 = await ethers.provider.getSigner(1)
> let contractAsSigner1 = coin.connect(signer1)
```

Now we can call the contract with `signer1`, which is `account[1]`.

Let's check balances now:

```
> value = await coin.balanceOf(accounts[0])
BigNumber { _hex: '0x075bccb6', _isBigNumber: true }
> value.toString()
'123456694'
> value = await coin.balanceOf(accounts[1])
BigNumber { _hex: '0x5f', _isBigNumber: true }
> value.toString()
'95'
```

We've successfully transferred 5 tokens from accounts[1] to accounts[0]

## Summary

Now you have the tools you need to launch a local Avalanche network, create a Hardhat project, as well as create, compile, deploy and interact with Solidity contracts.

Join our [Discord Server](#) to learn more and ask any questions you may have.

# Using Truffle with the Avalanche C-Chain

## Introduction

[Truffle Suite](#) is a toolkit for launching decentralized applications (dapps) on the EVM. With Truffle you can write and compile smart contracts, build artifacts, run migrations and interact with deployed contracts. This tutorial illustrates how Truffle can be used with Avalanche's C-Chain, which is an instance of the EVM.

## Requirements

You've completed [Run an Avalanche Node](#) and are familiar with [Avalanche's architecture](#). You've also performed a cross-chain swap via this [this tutorial](#) to get funds to your C-Chain address.

## Dependencies

- [Avalanche Network Runner](#) is a tool for running a local Avalanche network. It's similar to Truffle's [Ganache](#).
  - [NodeJS](#) v8.9.4 or later.
  - Truffle, which you can install with `npm install -g truffle`

## Start up a Local Avalanche Network

[Avalanche Network Runner](#) allows you to spin up private test network deployments. Start a local five node Avalanche network:

```
cd /path/to/avalanche-network-runner
start a five node staking network
.go run examples/local/fivenodenetwork/main.go
```

A five node Avalanche network is running on your machine. Network will run until you `Ctrl + C` to exit.

## Create Truffle Directory and Install Dependencies

Open a new terminal tab to so we can create a `truffle` directory and install some further dependencies.

First, navigate to the directory within which you intend to create your `truffle` working directory:

```
cd /path/to/directory
```

Create and enter a new directory named `truffle`:

```
mkdir truffle; cd truffle
```

Use `npm` to install [web3](#), which is a library through which we can talk to the EVM and [AvalancheJS](#) which is used for cross chain swaps.

```
npm install web3 avalanche -s
```

We'll use `web3` to set an HTTP Provider which is how `web3` will speak to the EVM. Lastly, create a boilerplate truffle project:

```
truffle init
```

The local development network pre-funds some static addresses when created. We'll use [@truffle/hdwallet-provider](#) to use these pre-funded addresses as our accounts.

```
npm install @truffle/hdwallet-provider
```

## Update `truffle-config.js`

One of the files created when you ran `truffle init` is `truffle-config.js`. Add the following to `truffle-config.js`.

```
const Web3 = require("web3")
const HDWalletProvider = require("@truffle/hdwallet-provider")

const protocol = "http"
const ip = "localhost"
const port = 9650
Web3.providers.HttpProvider.prototype.sendAsync =
 Web3.providers.HttpProvider.prototype.send
const provider = new Web3.providers.HttpProvider(
 `${protocol}://${ip}:${port}/ext/bc/C/rpc`
)

const privateKeys = [
 "0x56289e99c94b6912bf12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027",
 "0xb4198529994b0dc604278c99d153cf069d594753d471171a1d102a10438e07",
 "0x15614556be13730e9e8d6eacc1603143e7b96987429df8726384c2ec4502ef6e",
 "0x31b571bf6894a248831ff937bb49f7754509fe93bbd2517c9c73c4144c0e97dc",
 "0x6934bef917e01692b789da754a0eae31a8536eb465e7bfff752ea291dad88c675",
 "0xe700bdbdbc279b808b1ec45f8c2370e4616d3a02c336e68d85d4668e08f53cff",
 "0xbbc2865b76ba28016bc2255c7504d000e04ae01934b04c694592a6276988630",
 "0xcd8fd34f687ced8cc968854f8a99ae47712c4f4183b78dcc4a903d1fe8cbf60",
 "0x86f78c5416151fe3546dece84fda4b4b1e36089f2dbc48496faf3a950f16157c",
 "0x750839e9dbbd2a0910efe40f50b2f3b2f59f5580bb4b83bd8c1201cf9a010a",
]

module.exports = {
 networks: {
 development: {
 provider: () => {
 return new HDWalletProvider({
 privateKeys: privateKeys,
 providerOrUrl: provider,
 })
 },
 network_id: "*",
 gas: 3000000,
 gasPrice: 225000000000,
 },
 },
}
```

Note that you can change the `protocol`, `ip` and `port` if you want to direct API calls to a different AvalancheGo node. Also note that we're setting the `gasPrice` and `gas` to the appropriate values for the Avalanche C-Chain.

## Add Storage.sol

In the `contracts` directory add a new file called `Storage.sol` and add the following block of code:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.22 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

 uint256 number;

 /**
 * @dev Store value in variable
 * @param num value to store
 */
 function store(uint256 num) public {
 number = num;
 }

 /**
 * @dev Return value
 * @return value of 'number'
 */
 function retrieve() public view returns (uint256){
 return number;
 }
}
```

`Storage` is a solidity smart contract which lets us write a number to the blockchain via a `store` function and then read the number back from the blockchain via a `retrieve` function.

## Add New Migration

Create a new file in the `migrations` directory named `2_deploy_contracts.js`, and add the following block of code. This handles deploying the `Storage` smart contract to the blockchain.

```
const Storage = artifacts.require("Storage")

module.exports = function (deployer) {
 deployer.deploy(Storage)
}
```

## Compile Contracts with Truffle

Any time you make a change to `Storage.sol` you need to run `truffle compile`.

```
truffle compile
```

You should see:

```
Compiling your contracts...
=====
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/Storage.sol
> Artifacts written to /path/to/build/contracts
> Compiled successfully using:
 - solc: 0.5.16+commit.9c3226ce.Emscripten clang
```

## Accounts on C-Chain

When deploying smart contracts to the C-Chain, truffle will default to the first available account provided by your C-Chain client as the `from` address used during migrations. We have added some pre-defined private keys as our accounts in the `truffle-config.json`. The first and default account should have some pre-funded AVAX.

### Truffle Accounts

You can view imported accounts with truffle console.

To open the truffle console:

```
truffle console --network development
```

Note: If you see `Error: Invalid JSON RPC response: "API call rejected because chain is not done bootstrapping"`, you need to wait until network is bootstrapped and ready to use. It should not take too long.

Inside truffle console:

```
truffle(development)> accounts
[
 '0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC',
 '0x9632a79656af553F58738B0FB750320158495942',
 '0x55ee05dF718f1a5C1441e76190EB1a19eE2C9430',
 '0x4Cf2eD3665F6bFA95cE6A11CFdb7A2EF5FC1C7E4',
 '0xDb891D1b901D48750568962F86656083935C7A8',
 '0x01253b2E2EB0Fd64649FA468bF7b95ca933B2De',
 '0x78A23300E04FB5dD2820E23cc6797388b21fd5',
 '0x3C7daE394BF8e9EE1359ad14C1C47003bd0293',
 '0x61e0B3CD93F36847Abbd5d40d6F00a8eCf63cffB',
 '0x0Fa8E536Be85F32724D57A37758761B86416123'
```

You can see balances with:

```
truffle(development)> await web3.eth.getBalance(accounts[0])
'50000000000000000000000000000000'

truffle(development)> await web3.eth.getBalance(accounts[1])
'0'
```

Notice that `accounts[0]` (default account) has some balance, while `accounts[1]` has no balance

## Scripting Account Funding

There is a convenient script that funds the accounts list . You can find it [here](#). You can also download it using this command:

```
wget -nd -m https://raw.githubusercontent.com/ava-labs/avalanche-docs/master/scripts/fund-cchain-addresses.js
```

You can run the script with:

```
truffle exec fund-cchain-addresses.js --network development
```

Script will fund 1000 AVAX to each account in accounts list above. After successfully running the script you can check balances with:

```
truffle(development)> await web3.eth.getBalance(accounts[0]).
'50000000000000000000000000000000'
truffle(development)> await web3.eth.getBalance(accounts[1]).
'10000000000000000000000000000000'
```

## Fund Your Account

If you wish to fund accounts your own, follow the steps in this [tutorial](#). You'll need to send at least 135422040 nAVAX to the account to cover the cost of contract deployments.

## Personal APIs

Personal APIs interact with node's accounts. `web3` has some functions that uses it, e.g: `web3.eth.personal.newAccount`, `web3.eth.personal.unlockAccount` etc... However this API is disabled by default. It can be activated with `C-chain / Coreth` configs. The Avalanche Network Runner currently does not support activating this API. So if you want to use these features you need to run your own network manually with `internal-private-personal` API enabled via the `eth-apis` flag. See and [C-Chain Configs](#).

## Run Migrations

Now everything is in place to run migrations and deploy the storage contract:

```
truffle(development)> migrate --network development
```

You should see:

```
Compiling your contracts...
=====
> Everything is up to date, there is nothing to compile.

Migrations dry-run (simulation)
```

```
=====
> Network name: 'development-fork'
> Network id: 1
> Block gas limit: 99804786 (0x5f2e672)

1_initial_migration.js
=====

Deploying 'Migrations'

> block number: 4
> block timestamp: 1607734632
> account: 0x34Cb796d4D6A3e7F41c4465C65b9056Fe2D3B8FD
> balance: 1000.91683679
> gas used: 176943 (0x2b32f)
> gas price: 225 gwei
> value sent: 0 ETH
> total cost: 0.08316321 ETH

> Total cost: 0.08316321 ETH

2_deploy_contracts.js
=====

Deploying 'Storage'

> block number: 6
> block timestamp: 1607734633
> account: 0x34Cb796d4D6A3e7F41c4465C65b9056Fe2D3B8FD
> balance: 1000.8587791
> gas used: 96189 (0x177bd)
> gas price: 225 gwei
> value sent: 0 ETH
> total cost: 0.04520883 ETH

> Total cost: 0.04520883 ETH

Summary
=====
> Total deployments: 2
> Final cost: 0.13542204 ETH
```

If you didn't create an account on the C-Chain you'll see this error:

```
Error: Expected parameter 'from' not passed to function.
```

If you didn't fund the account, you'll see this error:

```
Error: *** Deployment Failed ***

"Migrations" could not deploy due to insufficient funds
 * Account: 0x090172CD36e9f4906Af17B2C36D662E69f162282
 * Balance: 0 wei
 * Message: sender doesn't have enough funds to send tx. The upfront cost is: 14100000000000000000000000000000 and the sender's account
only has: 0
 * Try:
 + Using an adequately funded account
```

## Interacting with Your Contract

Now the `Storage` contract has been deployed. Let's write a number to the blockchain and then read it back. Open the truffle console again:

Get an instance of the deployed `Storage` contract:

```
truffle(development)> let instance = await Storage.deployed()
```

This returns:

```
undefined
```

## Writing a number to the blockchain

Now that you have an instance of the `Storage` contract, call its `store` method and pass in a number to write to the blockchain.

```
truffle(development) > instance.store(1234)
```

You should see something like:

## Reading a Number From the Blockchain

To read the number from the blockchain, call the `retrieve` method of the `Storage` contract instance.

```
truffle(development)> let i = await instance.retrieve()
```

This should return:

The result of the call to `retrieve` is a `BN` (big number). Call its `toNumber` method to see the value:

```
truffle(development) > import('')
```

You should see the number you stored.

1234

## Summary

Now you have the tools you need to launch a local Avalanche network, create a truffle project, as well as create, compile, deploy and interact with Solidity contracts.

## Verifying Smart Contracts Using Hardhat and Snowtrace

This tutorial assumes that the contract was deployed using Hardhat and that all Hardhat dependencies are properly installed.

After deploying a smart contract one can verify the smart contract on Snowtrace in three steps:

1. Flatten the Smart Contract
  2. Clean up the flattened contract
  3. Verify using the Snowtrace GUI

## Flatten a Smart Contract using Hardhat

To flatten the contract run the command: `npx hardhat flatten <path-to-contract> >> <flat-contract-name>.sol`

## Clean up the flattened Smart Contract

Some clean-up may be necessary to get the code to compile properly in the Snowtrace Contract Verifier.

- Remove all but the top SPDX license.
    - If the contract uses multiple SPDX licenses, use both licenses by adding AND: `SPDX-License-Identifier: MIT AND BSD-3-Clause`

## Verify the Smart Contract using Snowtrace

1. Search for the contract in Snowtrace

2. Click on the contract tab

1. If the contract is unverified you will see something similar to the image below

3. Click Verify and Publish

4. On the next screen in the dropdown menus select the following

1. Solidity (Single file)

2. The compiler version you used to compile the deployed contract

3. The open-source license type (select none if applicable)

5. Copy and paste the code from the flattened contract into the appropriate box

6. If optimization was used when compiling the contract, make sure to select "Yes" in the dropdown menu labeled "Optimization"

1. If optimization was used, expand the bottom box labeled "Misc Settings" and input the number of runs

7. Select Verify and Publish

1. If successful, all Contracts with the same bytecode will be verified

2. If unsuccessful, read the error messages provided and make the appropriate changes

1. Ensure to check that the compiler version and optimizer runs are the same as when you compiled the contract prior to deployment

## Verifying with Hardhat-Verify

This part of the tutorial assumes that the contract was deployed using Hardhat and that all Hardhat dependencies are properly installed to include  
`@nomiclabs/hardhat-etherscan`.

You will need to create a `.env.json` with your *Wallet Seed Phrase* and *Snowtrace API key*

You will need to obtain an *API key* [here](#)

Example `.env.json`:

```
{
 "Mnemonic": "your-wallet-seed-phrase",
 "APIKey": "your-snowtrace-api-key"
}
```

Below is a sample `hardhat.config.ts` used for deployment and verification (See LN 45: `etherscan`)

```
import { task } from "hardhat/config"
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers"
import { BigNumber } from "ethers"
import "@typechain/hardhat"
import "@nomiclabs/hardhat-ethers"
import "@nomiclabs/hardhat-waffle"
import "hardhat-gas-reporter"
import "@nomiclabs/hardhat-etherscan"
import { MNEMONIC, APIKEY } from "./.env.json"

// When using the hardhat network, you may choose to fork Fuji or Avalanche Mainnet
// This will allow you to debug contracts using the hardhat network while keeping the current network state
// To enable forking, turn one of these booleans on, and then run your tasks/scripts using ``--network hardhat``
// For more information go to the hardhat guide
// https://hardhat.org/hardhat-network/
// https://hardhat.org/guides/mainnet-forking.html
const FORK_FUJI = false
const FORK_MAINNET = false
const forkingData = FORK_FUJI
? {
 url: "https://api.avax-test.network/ext/bc/C/rpc",
}
```

```

: FORK_MAINNET
? {
 url: "https://api.avax.network/ext/bc/C/rpc",
}
: undefined

task(
 "accounts",
 "Prints the list of accounts",
 async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners()
 accounts.forEach((account: SignerWithAddress): void => {
 console.log(account.address)
 })
 }
)

task(
 "balances",
 "Prints the list of AVAX account balances",
 async (args, hre): Promise<void> => {
 const accounts: SignerWithAddress[] = await hre.ethers.getSigners()
 for (const account of accounts) {
 const balance: BigNumber = await hre.ethers.provider.getBalance(
 account.address
)
 console.log(` ${account.address} has balance ${balance.toString()}`)
 }
 }
)
export default {
 etherscan: {
 // Your API key for Snowtrace
 // Obtain one at https://snowtrace.io/
 apiKey: APIKEY,
 },
 solidity: {
 compilers: [
 {
 version: "0.8.0",
 },
 {
 version: "0.8.10",
 },
],
 },
 networks: {
 hardhat: {
 gasPrice: 225000000000,
 chainId: 43114, //Only specify a chainId if we are not forking
 // forking: {
 // url: 'https://api.avax.network/ext/bc/C/rpc',
 // },
 },
 fuji: {
 url: "https://api.avax-test.network/ext/bc/C/rpc",
 gasPrice: 225000000000,
 chainId: 43113,
 accounts: { mnemonic: MNEMONIC },
 },
 mainnet: {
 url: "https://api.avax.network/ext/bc/C/rpc",
 gasPrice: 225000000000,
 chainId: 43114,
 accounts: { mnemonic: MNEMONIC },
 },
 },
}

```

Once the contract is deployed, verify with hardhat verify by running the following:

```
npx hardhat verify <contract address> <arguments> --network <network>
```

Example:

```
npx hardhat verify 0x3972c87769886C4f1FF3a8b52bc57738E82192D5 MockNFT Mock ipfs://QmQ2RFEmZaMds8bRjZCTJxo4DusvcBdLTS6XuDbhp5BZjY
100 --network fuji
```

You can also verify contracts programmatically via script

Example:

```
import console from "console"
const hre = require("hardhat")

// Define the NFT
const name = "MockNFT"
const symbol = "Mock"
const _metadataUri = "ipfs://QmQ2RFEmZaMds8bRjZCTJxo4DusvcBdLTS6XuDbhp5BZjY"
const _maxTokens = "100"

async function main() {
 await hre.run("verify:verify", {
 address: "0x3972c87769886C4f1FF3a8b52bc57738E82192D5",
 constructorArguments: [name, symbol, _metadataUri, _maxTokens],
 })
}

main()
.then(() => process.exit(0))
.catch((error) => {
 console.error(error)
 process.exit(1)
})
```

First create your script, then execute it via hardhat by running the following:

```
npx hardhat run scripts/<scriptname.ts> --network <network>
```

Example:

```
npx hardhat run scripts/5-verifyNFT.ts --network fuji
```

Verifying via terminal will not allow you to pass an array as an argument, however, you can do this when verifying via script by including the array in your *Constructor Arguments*

Example: (see LN13 `_custodians`, LN 30 `_custodians`)

```
import console from "console"
const hre = require("hardhat")

// Define the NFT
const name = "MockNFT"
const symbol = "Mock"
const _metadataUri =
 "ipfs://QmQn2jepp3jZ3tVxoCisMMF8kSi8c5uPKYxd7lxGwg38hv/Example"
const _royaltyRecipient = "0xcd3b766ccdd6ae721141f452c550ca635964ce71"
const _royaltyValue = "500000000000000000"
const _custodians = [
 "0x8626f6940e2eb28930efb4cef49b2d1f2c9c1199",
 "0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266",
 "0xdd2fd4581271e230360230f9337d5c0430bf44c0",
]
const _saleLength = "172800"
const _claimAddress = "0xcd3b766ccdd6ae721141f452c550ca635964ce71"
```

```

async function main() {
 await hre.run("verify:verify", {
 address: "0x08bf160B8e56899723f2E6F9780535241F145470",
 constructorArguments: [
 name,
 symbol,
 _metadataUri,
 _royaltyRecipient,
 _royaltyValue,
 _custodians,
 _saleLength,
 _claimAddress,
],
 })
}

main()
.then(() => process.exit(0))
.catch((error) => {
 console.error(error)
 process.exit(1)
})

```

## Verifying Smart Contracts with Truffle Verify

*This tutorial includes items from the [truffle quickstart docs](#)*

*Inspired by [truffle verify docs](#)*

### Create a Project

Make sure you have truffle installed:

```
npm install -g truffle
```

Create a new directory for your Truffle project:

```
mkdir MetaCoin
cd MetaCoin
```

Download ("unbox") the MetaCoin box:

```
truffle unbox metacoin
```

Once this operation is completed, you'll now have a project structure with the following items:

- `contracts/` : Directory for Solidity contracts
- `migrations/` : Directory for scriptable deployment files
- `test/` : Directory for test files for testing your application and contracts
- `truffle.js` : Truffle configuration file

### Compiling

Before we compile our smart contract, we must set up our environment

Run the following commands:

```
npm init -y
```

```
yarn add @truffle/hdwallet-provider yarn add -D truffle-plugin-verify
```

Create a `.env.json` file in your project's root directory:

```
{
 "mnemonic": "your-wallet-seed-phrase",
 "snowtraceApiKey": "your-snowtrace-api-key"
}
```

Get your Snowtrace API key [here](#)

Configure your `truffle-config.js` file to the appropriate settings:

```

/**
 * Use this file to configure your truffle project. It's seeded with some
 * common settings for different networks and features like migrations,
 * compilation and testing. Uncomment the ones you need or modify
 * them to suit your project as necessary.
 *
 * More information about configuration can be found at:
 *
 * trufflesuite.com/docs/advanced/configuration
 *
 * To deploy via Infura you'll need a wallet provider (like @truffle/hdwallet-provider)
 * to sign your transactions before they're sent to a remote public node. Infura accounts
 * are available for free at: infura.io/register.
 *
 * You'll also need a mnemonic - the twelve word phrase the wallet uses to generate
 * public/private key pairs. If you're publishing your code to GitHub make sure you load this
 * phrase from a file you've .gitignored so it doesn't accidentally become public.
 *
 */

const HDWalletProvider = require("@truffle/hdwallet-provider")

//
const { snowtraceApiKey, mnemonic } = require("./.env.json")

module.exports = {
 /**
 * Networks define how you connect to your ethereum client and let you set the
 * defaults web3 uses to send transactions. If you don't specify one truffle
 * will spin up a development blockchain for you on port 9545 when you
 * run `develop` or `test`. You can ask a truffle command to use a specific
 * network from the command line, e.g.
 *
 * $ truffle test --network <network-name>
 */

 plugins: ["truffle-plugin-verify"],
 api_keys: {
 snowtrace: snowtraceApiKey,
 },
 networks: {
 fuji: {
 provider: () =>
 new HDWalletProvider(
 mnemonic,
 `https://api.avax-test.network/ext/bc/C/rpc`
),
 network_id: 1,
 timeoutBlocks: 200,
 confirmations: 5,
 },
 },
}

```

*Network can be configured for Mainnet deployment(see Alternatives)*

Run the following command:

```
truffle compile
```

Once this operation is completed, your `./build/contracts` folder should contain the following items:

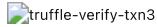
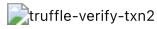
- ConvertLib.json
- MetaCoin.json
- Migrations.json

## Migrate

Run the following command:

```
npx truffle migrate --network fuji
```

You should see the TX activity in your terminal truffle-verify-txn1



## Truffle Verify

Truffle verify allows users to verify contracts from the CLI

### Fuji Testnet

Take a look at the Fuji Testnet Explorer [here](#) and read more about truffle verify [here](#)

If you have issues, contact us on [Discord](#)

1. Run the following command:

```
npx truffle run verify ConvertLib MetaCoin --network fuji
```

2. Wait for the verification message from the CLI

3. View the verified contract

### Mainnet

Configure your `truffle-config.js` file to the appropriate settings:

```
module.exports = {
...
 plugins: [
 'truffle-plugin-verify'
],
 api_keys: {
 snowtrace: snowtraceApiKey
 },
 networks: {

 mainnet: {
 provider: () => new HDWalletProvider(mnemonic, `https://api.avax.network/ext/bc/C/rpc`),
 network_id: 1,
 timeoutBlocks: 200,
 confirmations: 5
 }
 }
};
```

Run the following commands:

```
truffle migrate --network mainnet
```

```
truffle verify CovertLib MetaCoin --network mainnet
```

## Thanks for reading ▲

**description:** The purpose of this document is to help you with launching your existing Ethereum dapp on Avalanche, get the basics of Avalanche Platform and how it works.

## Launch Your Ethereum Dapp on Avalanche

### Overview

The purpose of this document is to help you with launching your existing dapp on Avalanche. It contains a series of resources designed to help you get the basics of Avalanche Platform and how it works, show how to connect to the network, how to use your existing tools and environments in developing and deploying on Avalanche, as well as some common pitfalls you need to consider when running your dapp on Avalanche.

### Platform Basics

Avalanche is a [network of networks](#). It means that it is not a single chain running a single, uniform type of blocks. It contains multiple Subnets, each running one of more heterogeneous chains. But, to run an Ethereum dapp on a low-fee, fast network with instant finality, we don't need to concern ourselves with that right now.

Using the link above you can find out more if you wish, but all you need to know right now is that one of the chains running on Avalanche Primary Network is the C-Chain (contract chain).

C-Chain runs a fork of [go-ethereum](#) called [coreth](#) that has the networking and consensus portions replaced with Avalanche equivalents. What's left is the Ethereum VM, which runs Solidity smart contracts and manages data structures and blocks on the chain. As a result, you get a blockchain that can run all the Solidity smart contracts from Ethereum, but with much greater transaction bandwidth and instant finality that [Avalanche's revolutionary consensus](#) enables.

Coreth is loaded as a plugin into [AvalancheGo](#), the client node application used to run Avalanche network.

As far as your dapp is concerned, it will be running the same as on Ethereum, just quicker and cheaper. Let's find out how.

## Accessing Avalanche C-Chain

C-Chain exposes the [same API](#) as [go-ethereum](#), so you can use all the familiar APIs that are available on Ethereum for interaction with the platform.

There are multiple ways of working with the C-Chain.

### Through Core

Powered by Avalanche, [Core](#) is an all-in-one operating system bringing together Avalanche apps, Subnets, bridges, and NFTs in one seamless, high-performance browser experience. Putting in another way, Core is more than a wallet. It is a curated web3 operating system combining Wallet, Explorer, Bridge, Subnets, dApps, and more.

In your application's web interface, follow [this to add Avalanche programmatically](#).

### Through MetaMask

You can access C-Chain through MetaMask, by defining a custom network. Go to MetaMask, log in, click the network dropdown, and select 'Custom RPC'. Data for Avalanche is as follows.

#### Avalanche Mainnet Settings:

- **Network Name:** Avalanche Mainnet C-Chain
- **New RPC URL:** <https://api.avax.network/ext/bc/C/rpc>
- **ChainID:** 43114
- **Symbol:** AVAX
- **Explorer:** <https://snowtrace.io/>

#### Fuji Testnet Settings:

- **Network Name:** Avalanche Fuji C-Chain
- **New RPC URL:** <https://api.avax-test.network/ext/bc/C/rpc>
- **ChainID:** 43113
- **Symbol:** AVAX
- **Explorer:** <https://testnet.snowtrace.io/>

In your application's web interface, you can [add Avalanche programmatically](#), so your users don't have to enter the network data manually. To see the adding custom network flow in action, check out [Pangolin DEX](#).

## Using the Public API Nodes

Instead of proxying network operations through MetaMask, you can use the public API, which consists of a number of AvalancheGo nodes behind a load balancer.

The C-Chain API endpoint is <https://api.avax.network/ext/bc/C/rpc> for the Mainnet and <https://api.avax-test.network/ext/bc/C/rpc> for the testnet.

For more information, see [documentation](#).

However, public API does not expose all the APIs that are available on the node, as some of them would not make sense on a publicly accessible service, and some would present a security risk. If you need to use an API that is not available publicly, you can run your own node.

## Running Your Own Node

If you don't want your dapp to depend on a centralized service you don't control, or have specific needs that cannot be met through the public API, you can run your own node and access the network that way. Running your own node also avoids potential issues with public API congestion and rate-limiting.

For development and experimental purposes, [here](#) is a tutorial that shows how to download, build, and install AvalancheGo. Simpler solution is to use the prebuilt binary, available on [GitHub](#). If you're going to run a node on a Linux machine, you can use the [installer script](#) to install the node as a `systemd` service. Script also handles node upgrading. If you want to run a node in a docker container, there are [build scripts](#) in the AvalancheGo repo for various Docker configs.

### Node Configuration

Node configuration options are explained [here](#). But unless you have specific needs, you can mostly leave the main node config options at their default values.

On the other hand, you will most likely need to adjust C-Chain configuration to suit your intended use. You can look up complete configuration options for C-Chain [here](#) as well as the default configuration. Note that only the options that are different from their default values need to be included in the config file.

By default, the C-Chain config file is located at `$HOME/.avalanchego/configs/chains/C/config.json`. We will go over how to adjust the config to cover some common use cases in the following sections.

#### Running an Archival Node

If you need Ethereum [Archive Node](#) functionality, you need to disable C-Chain pruning, which is enabled by default to conserve disk space. To preserve full historical state, include `"pruning-enabled": false` in the C-Chain config file.

:::note

After changing the flag to disable the database pruning, you will need to run the bootstrap process again, as the node will not backfill any already pruned and missing data.

To re-bootstrap the node, stop it, delete the database (by default stored in `~/.avalanche/go/db/`) and start the node again.

...

#### Running a Node in Debug Mode

By default, debug APIs are disabled. To enable them, you need to enable the appropriate EVM APIs in the config file by including the `eth-apis` value in your C-Chain config file to include the `debug`, `debug-tracer`, and `internal-debug` APIs.

:::note

Including the `eth-apis` in the config flag overrides the defaults, so you need to include the default APIs as well!

...

#### Example C-Chain Config File

An example C-Chain config file that includes the archival mode, enables debug APIs as well as default EVM APIs:

```
{
 "eth-apis": [
 "eth",
 "eth-filter",
 "net",
 "web3",
 "internal-eth",
 "internal-blockchain",
 "internal-transaction",
 "debug-tracer"
,
 "pruning-enabled": false
}
```

Default config values for the C-Chain can be seen [here](#).

#### Running a Local Test Network

If you need a private test network to test your dapp, [Avalanche Network Runner](#) is a shell client for launching local Avalanche networks, similar to Ganache on Ethereum.

For more information, see [documentation](#).

## Developing and Deploying Contracts

Being an Ethereum-compatible blockchain, all of the usual Ethereum developer tools and environments can be used to develop and deploy dapps for Avalanche's C-Chain.

#### Remix

There is a [tutorial](#) for using Remix to deploy smart contracts on Avalanche. It relies on MetaMask for access to the Avalanche network.

#### Truffle

You can also use Truffle to test and deploy smart contracts on Avalanche. Find out how in this [tutorial](#).

#### Hardhat

Hardhat is the newest development and testing environment for Solidity smart contracts, and the one our developers use the most. Due to its superb testing support, it is the recommended way of developing for Avalanche.

For more information see [this doc](#).

## Avalanche Explorer

An essential part of the smart contract development environment is the explorer, which indexes and serves blockchain data. Mainnet C-Chain explorer is available at <https://snowtrace.io/> and testnet explorer at <https://testnet.snowtrace.io/>. Besides the web interface, it also exposes the standard [Ethereum JSON RPC API](#).

## Avalanche Faucet

For development purposes, you will need test tokens. Avalanche has a [Faucet](#) that drips test tokens to the address of your choice. Paste your C-Chain address there.

If you need, you can also run a faucet locally, but building it from the [repository](#).

## Contract Verification

Smart contract verification provides transparency for users interacting with smart contracts by publishing the source code, allowing everyone to attest that it really does what it claims to do. You can verify your smart contracts using the [C-Chain explorer](#). The procedure is simple:

- navigate to your published contract address on the explorer
- on the `code` tab select `verify & publish`
- copy and paste the flattened source code and enter all the build parameters exactly as they are on the published contract
- click `verify & publish`

If successful, the `code` tab will now have a green checkmark, and your users will be able to verify the contents of your contract. This is a strong positive signal that your users can trust your contracts, and it is strongly recommended for all production contracts.

See [this](#) for a detailed tutorial with Truffle.

## Contract Security Checks

Due to the nature of distributed apps, it is very hard to fix bugs once the application is deployed. Because of that, making sure your app is running correctly and securely before deployment is of great importance. Contract security reviews are done by specialized companies and services. They can be very expensive, which might be out of reach for single developers and startups. But, there are also automated services and programs that are free to use.

Most popular are:

- [Slither](#), here's a [tutorial](#)
- [MythX](#)
- [Mythril](#)

We highly recommend using at least one of them if professional contract security review is not possible. A more comprehensive look into secure development practices can be found [here](#).

## Gotchas and Things to Look out For

Avalanche Platform's C-Chain is EVM-compatible, but it is not identical. There are some differences you need to be aware of, otherwise, you may create subtle bugs or inconsistencies in how your dapps behave.

Here are the main differences you should be aware of.

### Measuring Time

Avalanche does not use the same mechanism to measure time as Ethereum which uses consistent block times. Instead, Avalanche supports asynchronous block issuance, block production targets a rate of every 2 seconds. If there is sufficient demand, a block can be produced earlier. If there is no demand, a block will not be produced until there are transactions for the network to process.

Because of that, you should not measure the passage of time by the number of blocks that are produced. The results will not be accurate, and your contract may be manipulated by third parties.

Instead of block rate, you should measure time simply by reading the timestamp attribute of the produced blocks. Timestamps are guaranteed to be monotonically increasing and to be within 30 seconds of the real time.

### Finality

On Ethereum, the blockchain can be reorganized and blocks can be orphaned, so you cannot rely on the fact that a block has been accepted until it is several blocks further from the tip (usually, it is presumed that blocks 6 places deep are safe). That is not the case on Avalanche. Blocks are either accepted or rejected within a second or two. And once the block has been accepted, it is final, and cannot be replaced, dropped, or modified. So the concept of 'number of confirmations' on Avalanche is not used. As soon as a block is accepted and available in the explorer, it is final.

### Using `eth_newFilter` and Related Calls with the Public API

If you're using the `eth_newFilter` API method on the public API server, it may not behave as you expect because the public API is actually several nodes behind a load balancer. If you make an `eth_newFilter` call, subsequent calls to `eth_getFilterChanges` may not end up on the same node as the first call, and you will end up with undefined results.

If you need the log filtering functionality, you should use a websocket connection, which ensures that your client is always talking to the same node behind the load balancer. Alternatively, you can use `eth_getLogs`, or run your own node and make API calls to it.

### Support

Using this tutorial you should be able to quickly get up to speed on Avalanche, deploy, and test your dapps. If you have questions, problems, or just want to chat with us, you can reach us on our public [Discord](#) server. We'd love to hear from you and find out what you're building on Avalanche!

## Non Fungible Tokens (NFTs) Overview

| Title                                                        | Description                                    |
|--------------------------------------------------------------|------------------------------------------------|
| <a href="#">Introduction to ERC721 (NFT) Smart Contracts</a> | Introduction to ERC721 (NFT) Smart Contracts.  |
| <a href="#">Preparing NFT Files</a>                          | Preparing image and metadata files for upload. |

## Introduction to ERC-721 (NFT) Smart Contracts

This tutorial will start you with a basic [ERC-721\(NFT\)](#) smart contract on the Avalanche Network, regardless of your previous development experience. We'll deploy our NFT on the Avalanche Fuji Testnet and view it on the Snowtrace Testnet Explorer. Note that these aren't transferable to the Mainnet. However, once you feel comfortable launching your project, you can do so on Avalanche Mainnet and list it on an NFT marketplace.

The following tools will be used during this tutorial:

- [Pinata](#): To store your NFT images and metadata.
- [OpenZeppelin's Wizard](#): to create the ERC-721 smart contract.
- [Remix IDE](#): To edit the code and deploy it to Fuji.
- [Avalanche Testnet Faucet](#): To fund the deployment.
- [MetaMask browser Extension](#): To process transactions related to funding and deploying the smart contract.
- [Snowtrace Testnet Explorer](#): To view the deployed smart contract.

:::caution

DISCLAIMER: This Solidity smart contract tutorial is for demonstration purposes only. Users should consider proper precautions, error handling, and safeguards for production use. No one at Ava Labs is responsible for your development, and you must take full responsibility for ensuring your code is secure.

:::

## Preparing Your NFT Files

The first step of setting up an NFT smart contract is having your NFT files ready to use. In this example, the files will get uploaded to Pinata, a pinning service that prevents files from being garbage collected on IPFS.

If you're unfamiliar with the process of uploading image and metadata files to an IPFS provider for NFT collection usage, please check out [this article on preparing NFT files](#). Ensure that your files are uploaded and your base URI is ready to plug into your smart contract.

Once the image and metadata files are ready, we can prepare to deploy a smart contract.

## Preparing Your Environment

### MetaMask Extension

You'll need the MetaMask Extension installed on whatever browser you're using to be able to fund the deployment of the smart contract. If you've not done so already, download MetaMask and [add the Fuji network to MetaMask](#). Create or import a Fuji account as necessary.

### Getting Testnet Funds

Because we're deploying on the Fuji Network, you'll need to get AVAX on the Fuji network. If you visit the [Avalanche Faucet](#), you can request up to 2 Fuji AVAX per day. Please enter the C Chain address of the account linked to your MetaMask in the previous step.



## Creating the Smart Contract

To create the smart contract, we're going to use [Open Zeppelin](#). Open Zeppelin is a key tool for building smart contracts quickly and easily. While we're only scratching the surface in this tutorial, ample documentation is available on their website for you to read when you want to build more complex contracts.

Open Zeppelin provides a [Contract Wizard](#) that will build out ERC contracts. To avoid any complex coding environments, we'll use this to create our ERC-721 contract.



Select `ERC-721` on the Contract Wizard to get started. This will create the contract in the [Solidity programming language](#).

As you can see, the template contract is bare-boned. We'll fill out the information in the left panel to auto-populate it into our contract. Make sure you change to the `ERC-721` tab as you get started to make the proper contract.

The Wizard auto-fills in a name and symbol for your NFT collection, which we'll modify here. I'm naming it `Photography` and giving it the symbol `FOTO`. If you chose your own files to use during this tutorial, you can choose a relevant name and symbol for your collection.

The Base URI field listed here is the URL of the metadata folder uploaded to Pinata(for example, ours is <https://gateway.pinata.cloud/ipfs/QmYdWxbiwsfsYcWlCYQPgYujAc9FMLPG3fgFcxFsksBssFa>). Paste that into the Base URI field. After the Wizard adds our variables to the template, our contract should look like this:



Next, we'll want to check the `Mintable` and `Auto Increment Ids` boxes. This will populate a mint function into our template that would handle the incrementing of token IDs on mint if we had more than one NFT in our collection. We still want it to auto-assign our 1 NFT, so we'll check it.

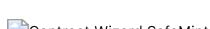
This automatically checks the `Ownable` button, which gives the `safeMint` function the `onlyOwner` modifier. This modifier indicates that only the owner of the smart contract will be able to successfully call the function.

:::note

This modifier should be removed when creating a smart contract for a public mint. Otherwise, users wouldn't be able to successfully mint the NFTs when calling the `safeMint` function. This tutorial only handles the owner's wallet address, so it is being left in.

:::

Now, our contract is a little more populated:



For this simple example, we'll not add any additional functionality to the `safeMint` function. Currently, it mints one NFT to the address specified in the function call. There is no cost to mint the NFT other than the gas fee for the transaction itself.

This `safeMint` function currently doubles as an airdrop function because the address the NFT is minted to does not need to be the function owner. This functionality becomes very useful when NFT collection owners want to give away NFTs for free outside of the normal minting window.

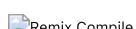
At this point, our smart contract is ready. At the top, you can click `Open in Remix` to get ready to deploy your smart contract.



## Deploying the Smart Contract with Remix

[Remix IDE](#) is a solidity compiler that allows you to edit, compile, and deploy your smart contract. This will prevent you from needing to download any other coding environments at this stage.

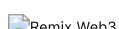
Once you've imported your contract, the first thing you need to do is compile it. Hit the `Compile` button on the left-hand side. You could also use the keyboard shortcut `Ctrl / Command + S`.



Once completed, you'll get a green checkmark on the far left tab and will see options to Publish on IPFS or Swarm. Those aren't important to our tutorial. Next, you'll click on the bottom tab on the left-hand side to move to the deployment page.



Now, we need to change the environment that Remix will try to use to deploy the smart contract. Click on the `Environment` drop-down, and select `Injected web3`.



This should prompt you to connect with your MetaMask account. Once connected, you can verify the correct connection by checking that the Account number matches your MetaMask address.



Now click on the `Contract` drop-down and select the contract you created and compiled. It should show up with the name you gave it in the Open Zeppelin Wizard.



Now, click deploy. This will open MetaMask and ask you to confirm the transaction. Click `Confirm`.

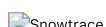


It may take a second, but once completed, your newly deployed contract will appear underneath the `Transactions Recorded` field.



Copy your contract's address and open the [Snowtrace Testnet Explorer](#). Paste your contract address in the search bar, and click `Search`.

You'll now see [your contract information on Snowtrace](#). The first transaction you see should be the contract deployment you just did in the Remix IDE.



## Minting an NFT

Now that you've deployed the contract, you can mint the NFT. Go back to the Remix IDE tab and click on your contract to expand its information. A list of functions will appear that you can interact with.



The only function you're interested in is the `safeMint` function. Click the drop-down arrow for the function to expand the address field.



Now, copy your MetaMask address and paste it into this address field. This will send the NFT to your address when the mint function is called. After, hit `transact`.

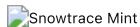
This will reopen MetaMask and ask you to verify the transaction. Click `Confirm` to mint your NFT.



Once the transaction has been confirmed, you'll see a green checkmark in the terminal at the bottom of the Remix IDE.



Head back to the Snowtrace Testnet explorer page for your contract and refresh it. You should now see a second transaction, your call to `safeMint`.



By clicking on the [TX Hash](#), you see that your NFT was created!



## Mainnet

All of the above steps can be used on Mainnet except the following changes:

- Make sure that you [connect to the Mainnet with MetaMask](#).
- Make sure that you have AVAX tokens in your account to cover transaction costs.
- You should use the Mainnet version of [Snowtrace Explorer](#) to view transactions.

## Preparing NFT Files for an ERC721 Smart Contract

The first step of setting up an NFT smart contract is having your NFT files ready to use. In this example, the files will get uploaded to [Pinata](#), a pinning service that prevents files from being garbage collected on IPFS. If you don't already have an account, please create one.

### Preparing the Images

This tutorial will create only 1 NFT, however, if you're interested in creating more, you're more than welcome to do so. The image I'm using is linked here if you'd like to use it.



Place your image file in a folder on your computer. Name this image `0`, so it'll be the first image pulled from the smart contract. It'll be the first (and only) NFT in this collection, however, if you're adding more images you'd continue naming them in sequential numeric order. You'll upload this folder to Pinata once your images are organized and named correctly.

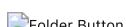
:::info

NOTE: Some projects start file names with `0`, and others with `1`. That choice will need to be consistent with the smart contract code. To be consistent with [this ERC-721 tutorial](#), we'll name this file `0`.

:::



After you log into Pinata, you'll see your dashboard. You'll see the upload button on the left. Click `Upload`, and then `Folder`.



You'll then select the folder that the image is in. You may get a pop-up from your browser confirming you want to upload the folder and the files in it. If you do, confirm by clicking `Upload`.



You'll then be prompted to name the folder you've uploaded. This is beneficial when you have several sets of folders uploaded to Pinata and are trying to keep them organized. After giving it a name, click `Upload` and wait for your file to upload. The quantity and size of the images could affect the upload time, but if you're starting small, it should only take a few seconds.

Once the upload is complete, you'll see your folder in your dashboard.



If you click on the folder name, it'll redirect you to the Pinata gateway to be able to view your newly uploaded files. If you have a paid Pinata account, it'll open the folder through your own gateway. Having a paid plan and personal gateway is NOT required for this tutorial but is recommended to have for larger collection sizes and hosting multiple folders.

If you right-click on the image, you can copy the image's url. This URL is important. Copy this down to use in the next step as we set up the metadata. For this example, my URL is <https://gateway.pinata.cloud/ipfs/QmPWbixyMsaNkR9v612bBFbncKGmgXDKz9CgMtDDD7Bymw/0.png>

## Preparing the Metadata

Now that we have the image uploaded and its URL, we can create the matching metadata file for it.

Where this NFT is going to be an ERC-721, we know we can use metadata standards often found on Marketplaces such as [Joepegs.com](#). The .json file below is an example of what the [metadata](#) should look like.

```
{
 "name": "",
 "tokenId": 0,
 "image": "",
 "description": "",
 "attributes": []
}
```

Now, we'll populate the values into the metadata file. You can choose any `name` and `description` that you want.

The `tokenId` here will be `0` so that it corresponds to the image we just uploaded. If uploading multiple files, this needs to be incremented in each file.

The `image` link is the URL we saved from the last step of the previous section. Paste that link here so the smart contract knows where to find the image file for your NFT. If uploading multiple files, the end of the URL (the specific image) needs to increment in each file.

The `attributes` field isn't quite as important here, but if you were uploading NFTs with several layers, the attributes would be the information of those specific layers. This is often used when calculating the rarity of NFTs to be able to rank them by how frequently their layers appear throughout the entire collection. It'll be erased in this tutorial.

Below is an example of how you'd fill out the fields in the metadata file.

```
{
 "name": "Cool Photography",
 "tokenId": 0,
 "image": "https://gateway.pinata.cloud/ipfs/QmPWbixyMsaNkR9v612bBFbncKGmgXDKz9CgMtDDD7Bymw/0.png",
 "description": "A cool image"
}
```

When saving this file, you want it to share the same name as the image it corresponds to. In this case, it is `0`.

Once the metadata file is uploaded to Pinata, the file extension will actually not be needed. It'll search for the file as a directory and be able to pull its information from there. To remove the file extension, follow these steps for a [Mac](#) environment, or these for a [Windows](#) environment.

Now that the file extension has been removed, place it in another folder as you did with the image file. They need to be SEPARATE folders.



You'll now repeat the folder upload process to add the metadata to Pinata. Follow the same steps as above. Once completed, you'll have both folders available on your dashboard.



Click on the metadata folder to be directed to the IPFS gateway and save the URL. This URL will be your base URL and won't need the direct file links. The smart contract will append the necessary file information for each NFT as needed. For example, my URL is <https://gateway.pinata.cloud/ipfs/QmYdWxbiwsfsYcW1CYQPgYujAc9FMLPG3fgFcxFsksbSsFa>.

Now that the image and metadata files are ready, we can prepare to deploy a smart contract by following this [ERC-721 tutorial](#).

## Smart Contracts Overview

| Title                                                                         | Description                                                                  |
|-------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| <a href="#">Create an ERC-20 Token Using Solidity</a>                         | Create an Ethereum ERC-20 token on Avalanche.                                |
| <a href="#">Deploy a Smart Contract on Avalanche Using Remix and MetaMask</a> | Deploy and test a smart contract on Avalanche using Remix and MetaMask.      |
| <a href="#">Verify Smart Contracts on the C-Chain Explorer</a>                | Verify Smart Contracts on the C-Chain Explorer                               |
| <a href="#">Add Avalanche Network Programmatically</a>                        | Simple tutorial showing how to add Avalanche network to your front-end code. |

## Add Avalanche Network Programmatically

This document shows how to integrate Avalanche Network with your Dapp, either by Core or MetaMask.

### Core

Powered by Avalanche, [Core](#) is an all-in-one operating system bringing together Avalanche apps, Subnets, bridges, and NFTs in one seamless, high-performance browser experience. Putting in another way, Core is more than a wallet. It is a curated web3 operating system combining Wallet, Explorer, Bridge, Subnets, dApps, and more.

Getting a Dapp ready to connect to Core is made simple with pre-built tools from the Core Team.

First download and install the Core browser extension from [here](#).

[avalanche-dapp-sdks](#) contains an example of how to connect via @web3-react/core to the Core extension specifically.

```
git clone https://github.com/ava-labs/avalanche-dapp-sdks.git
cd avalanche-dapp-sdks
yarn bootstrap
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanche-dapp-sdks.git
```

You can find more about SSH and how to use it [here](#). :::

Then check out [this sample project under packages/avalanche-connector-example](#)

```
cd packages/avalanche-connector-example
npm start
```

If everything works as expected, you should be able to load "<http://localhost:3000/>" on your browser, click on the "Connect Avalanche" button on the page as below:



Check out the [README](#) to see details how this works and use it to fit your needs.

This [Google Drive](#) has the assets needed to create a Connect with Core button.

## MetaMask

Adding new networks to MetaMask is not a trivial task for people that are not technically savvy, and it can be error prone. To help easier onboarding of users to your application it is useful to simplify that process as much as possible. This tutorial will show how to build a simple button in your front-end application that will automate the process of adding the Avalanche network to MetaMask.

### EIP-3035 & MetaMask

[EIP-3035](#) is an [Ethereum Improvement Proposal](#) that defines an RPC method for adding Ethereum-compatible chains to wallet applications.

Since March 2021 MetaMask has implemented that EIP as part of their MetaMask [Custom Networks API](#).

Let's see how it works.

### Data Structures

To add the Avalanche network to MetaMask, we need to prepare the data structures that will be contain all the necessary data.

Main network data:

```
export const AVALANCHE_MAINNET_PARAMS = {
 chainId: "0xA86A",
 chainName: "Avalanche Mainnet C-Chain",
 nativeCurrency: {
 name: "Avalanche",
 symbol: "AVAX",
 decimals: 18,
 },
 rpcUrls: ["https://api.avax.network/ext/bc/C/rpc"],
 blockExplorerUrls: ["https://snowtrace.io/"],
}
```

Test network data:

```
export const AVALANCHE_TESTNET_PARAMS = {
 chainId: "0xA869",
 chainName: "Avalanche Testnet C-Chain",
 nativeCurrency: {
 name: "Avalanche",
 symbol: "AVAX",
 decimals: 18,
 },
 rpcUrls: ["https://api.avax-test.network/ext/bc/C/rpc"],
```

```
 blockExplorerUrls: ["https://testnet.snowtrace.io/"],
}
```

## Adding the Network

To add the network to MetaMask, we need to call the `wallet_addEthereumChain` method, exposed by the web3 provider.

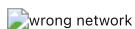
```
function addAvalancheNetwork() {
 injected.getProvider().then((provider) => {
 provider
 .request({
 method: "wallet_addEthereumChain",
 params: [AVALANCHE_MAINNET_PARAMS],
 })
 .catch((error: any) => {
 console.log(error)
 })
 })
}
```

Where `injected` is initialized as a `web3-react/injected-connector` used to interface with MetaMask APIs. Usage for other popular web frameworks is similar. Replace `AVALANCHE_MAINNET_PARAMS` with `AVALANCHE_TESTNET_PARAMS` if you want to add the test network.

Typical usage pattern would be to expose a button calling that method if you get `Wrong Network` or `Error connecting` errors when attempting to establish a connection to MetaMask.

## User Experience

When users first come to your dapp's website they need to approve connection to MetaMask. After they do that, if you don't detect successful web3 network connection, you can present them with a dialog asking them to confirm switch to a new network:



If they press the button, they are shown a dialog from MetaMask asking for approval to add the new network:



If they approve, your app will be connected to the Avalanche network. Very easy, no need for any data entry, no chance of wrong data entry. And that's it, users are ready to interact with your dapp!

## Conclusion

Dapps users are often not very technically sophisticated and onboarding them needs to be as seamless and easy as possible. Manually adding a new network is a hurdle than a certain percentage of your potential users will not be able to clear. Removing that requirement is a simple step that will enhance their experience and enable more users to get to actually use your dapp.

**If you have any questions, problems, or ideas on how to improve, or simply want to join our developer community, you can contact us on our [Discord](#) server.**

**description: ERC-20 tokens are the most fundamental and essential concept in Ethereum. This same token standard is adopted in the Avalanche ecosystem.**

## Create an ERC-20 Token Using Solidity

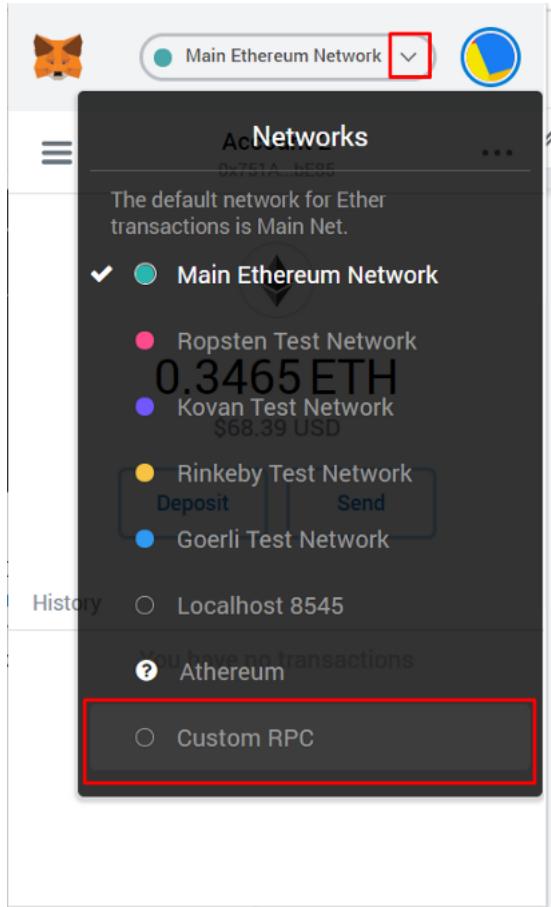
[ERC-20 tokens](#) are the most fundamental and essential concept in Ethereum. As the Avalanche community and the ecosystem are growing, new use cases and projects that are running on Ethereum or different chains would be implemented to Avalanche.

Therefore, we will be creating our own mintable ERC-20 token and will mint it to any address we want. The token will be generated on Avalanche C-Chain and will be accessible on that chain. We are using Fuji Testnet in this tutorial.

The article focuses on deploying a smart contract written with Solidity to Avalanche. This is the feature that Avalanche provides us - to be able to deploy any smart contract to the chain and no requirement for a new language specific contract concept to interact. Let's look at how to create an ERC-20 contract and deploy it to avalanche C-Chain.

## Set up MetaMask

The first thing we should set is a MetaMask wallet.



Click to MetaMask icon on the browser and select the network drop-down menu. Here we should connect to C-Chain. Click to "Custom RPC".

## Networks

Add Network

- Ethereum Mainnet 🔒
- Ropsten Test Network 🔒
- Rinkeby Test Network 🔒
- Goerli Test Network 🔒
- Kovan Test Network 🔒
- Localhost 8545
- New Network**

A malicious Ethereum network provider can lie about the state of the blockchain and record your network activity. Only add custom networks you trust.

Network Name

New RPC URL

Chain ID ⓘ

Symbol (optional)

Block Explorer URL (optional)

Cancel

Save

Now, we need to set these boxes with correct values.

- **Network Name:** Avalanche C-Chain
- **New RPC URL:** <https://api.avax-test.network/ext/bc/C/rpc>
- **ChainId:** 43113
- **Symbol:** AVAX
- **Explorer:** <https://testnet.snowtrace.io>

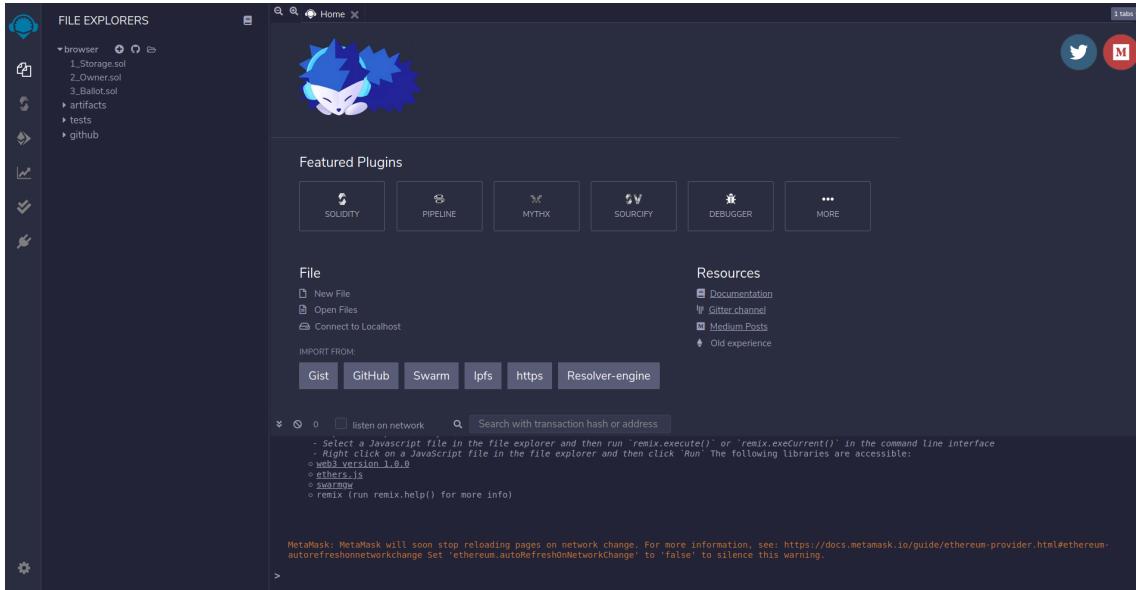
After setting up all the parameters correctly, we should see this page. For now, we have 0 AVAX.

## Fund Your C-Chain Address

For funding on the Fuji Testnet, you can use the Test Network Faucet. Navigate to <https://faucet.avax.network/> and paste your C-Chain address.

## Create Mintable Token

Now, we can create our mintable token on Remix. Open Remix on your browser or go to [this link](#).



You should view this page. On this page, first, click "SOLIDITY" from "Featured Plugins" and then click the "New File" button. When you click the New File button, you will see a pop-up that requires a file name. You can choose a name or leave the default.

Since we will use an ERC-20 contract from [OpenZeppelin](#), just paste this line to the file and save.

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/presets/ERC20PresetMinterPauser.sol";
```

The screenshot shows the Remix IDE after saving the file. The sidebar now includes "OpenZeppelin" under "github". The code editor shows the imported contract code. The status bar at the bottom indicates "94" changes.

After saving the file, we will see a bunch of files that are imported to remix. This is a remix feature that allows us to import a GitHub contract repository to remix by just giving the URL-link with an import statement.

The screenshot shows the Truffle UI interface. On the left, the 'FILE EXPLORERS' panel displays a tree view of Solidity files, including 'browser', 'Storage.sol', 'Owner.sol', 'Balloon.sol', 'artifacts', 'tests', 'github', 'OpenZeppelin', 'openzeppelin-contracts', 'contracts', 'presets', and 'token'. A file named 'ERC20PresetMinterPauser.sol' is selected in the 'presets' folder. On the right, the code editor displays the Solidity source code for 'ERC20PresetMinterPauser.sol'. The code defines a contract that inherits from Context, AccessControl, ERC20Burnable, and ERC20Pausable. It includes roles for minter and pauser, and a constructor that sets up these roles.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import "../access/AccessControl.sol";
import "../GSN/Context.sol";
import "../token/ERC20/ERC20.sol";
import "../token/ERC20/ERC20Burnable.sol";
import "../token/ERC20/ERC20Pausable.sol";

/**
 * @dev {ERC20} token, including:
 *
 * - ability for holders to burn (destroy) their tokens
 * - a minter role that allows for token minting (creation)
 * - a pauser role that allows to stop all token transfers
 *
 * This contract uses {AccessControl} to lock permissioned functions using the
 * different roles - head to its documentation for details.
 *
 * The account that deploys the contract will be granted the minter and pauser
 * roles, as well as the default admin role, which will let it grant both minter
 * and pauser roles to other accounts.
 */
contract ERC20PresetMinterPauser is Context, AccessControl, ERC20Burnable, ERC20Pausable {
 bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
 bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

 /**
 * @dev Grants `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE` and `PAUSER_ROLE` to the
 * account that deploys the contract.
 *
 * See {ERC20-constructor}.
 */
 constructor(string memory name, string memory symbol) public ERC20(name, symbol) {
 _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
 _setupRole(MINTER_ROLE, _msgSender());
 _setupRole(PAUSER_ROLE, _msgSender());
 }

 /**
 * @dev Creates `amount` new tokens for `to`.
 *
 * See {ERC20-_mint}.
 *
 * Requirements:
 *
 * - the caller must have the `MINTER_ROLE`.
 */
}

```

We have ERC20PresetMinterPauser.sol file in the presets. This file is written by OpenZeppelin according to ERC20 standards with minter functionality. After deploying this file, we will be the owner of the contract and thus have the authority and ability to mint the tokens.

The screenshot shows the Truffle UI interface with the 'SOLIDITY COMPILER' tab selected. On the left, the compiler configuration is set to version 0.6.6-commit.6c089d02, language Solidity, and EVM version compiler default. Under 'COMPILER CONFIGURATION', 'Auto compile' is checked. In the center, the code editor shows the same Solidity source code as in the previous screenshot. At the bottom, there are buttons for 'Compile' (highlighted in blue), 'Contract' (set to 'Context (Context.sol)'), and options to 'Publish on Swarm' or 'Publish on Ipfss'. At the very bottom, there are buttons for 'ABI' and 'Bytecode'.

## Deploy the Contract

Open the tab with label `Solidity compiler` and select the solidity version that matches with the solidity version written in file as `pragma solidity ....`. The version should be equal to or higher than the file's version. For example, in my file, `pragma solidity ^0.6.0` is written so the required version is 0.6.0 or higher. As shown, in the compiler the solidity version is 0.6.6, which is OK. After checking the solidity version click the compile button. If you did not change anything in the file, or the solidity version is not wrong, the contract should compile without any errors.

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

import "../access/AccessControl.sol";
import "../GSN/Context.sol";
import "../token/ERC20/ERC20Burnable.sol";
import "../token/ERC20/ERC20Pausable.sol";
import "../token/ERC20/ERC20Pausable.sol";

/**
 * @dev (ERC20) token, including:
 *
 * - ability for holders to burn (destroy) their tokens
 * - a minter role that allows for token minting (creation)
 * - a pauser role that allows to stop all token transfers
 *
 * This contract uses {AccessControl} to lock permissioned functions using the
 * different roles - head to its documentation for details.
 *
 * The account that deploys the contract will be granted the minter and pauser
 * roles, as well as the default admin role, which will let it grant both minter
 * and pauser roles to other accounts.
 */
contract ERC20PresetMinterPauser is Context, AccessControl, ERC20Burnable, ERC20Pausable {
 bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
 bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");

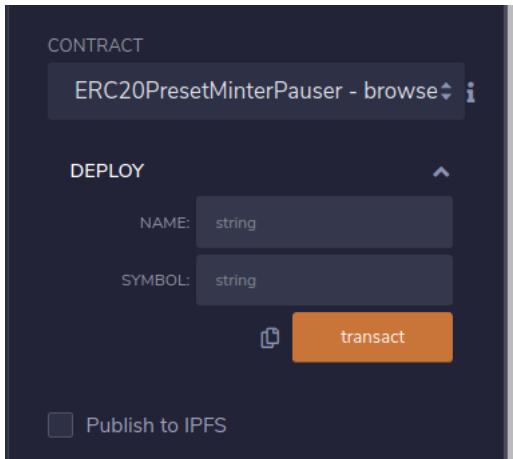
 /**
 * @dev Grants 'DEFAULT_ADMIN_ROLE', 'MINTER_ROLE' and 'PAUSER_ROLE' to the
 * account that deploys the contract.
 *
 * See {ERC20-constructor}.
 */
 constructor(string memory name, string memory symbol) public ERC20(name, symbol) {
 _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
 _setupRole(MINTER_ROLE, _msgSender());
 _setupRole(PAUSER_ROLE, _msgSender());
 }

 /**
 * @dev Creates 'amount' new tokens for 'to'.
 *
 * See {ERC20-_mint}.
 *
 * Requirements:
 *
 * - the caller must have the 'MINTER_ROLE'.
 */
}

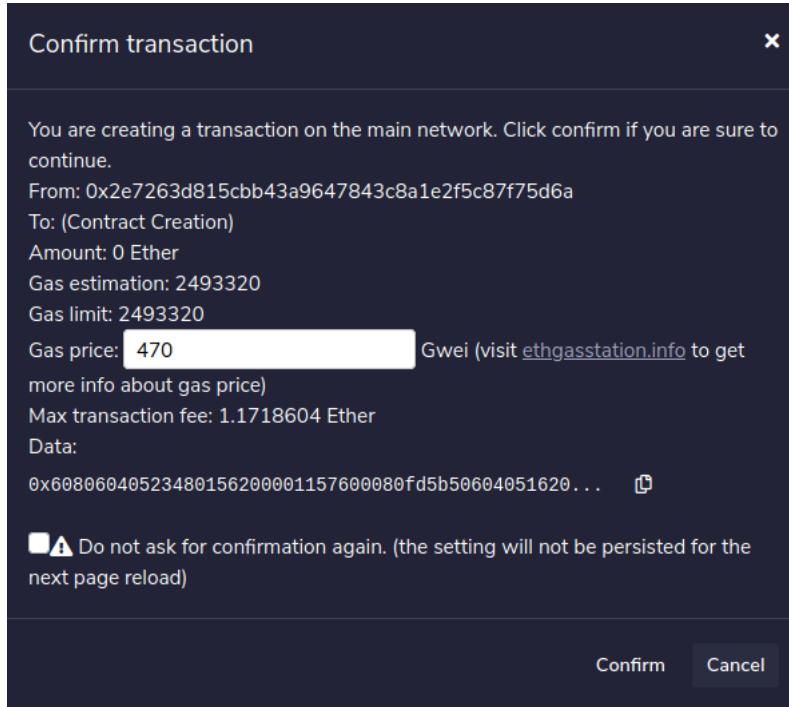
```

Then, let's jump to the tab with label `Deploy & run transactions`. Here before deploying our contract, we should change the environment. Click to the environment and select "Injected Web3." If a pop-up shows up and asks you to connect the account, click to connect. After, you should see the account address in the "ACCOUNT" text box.

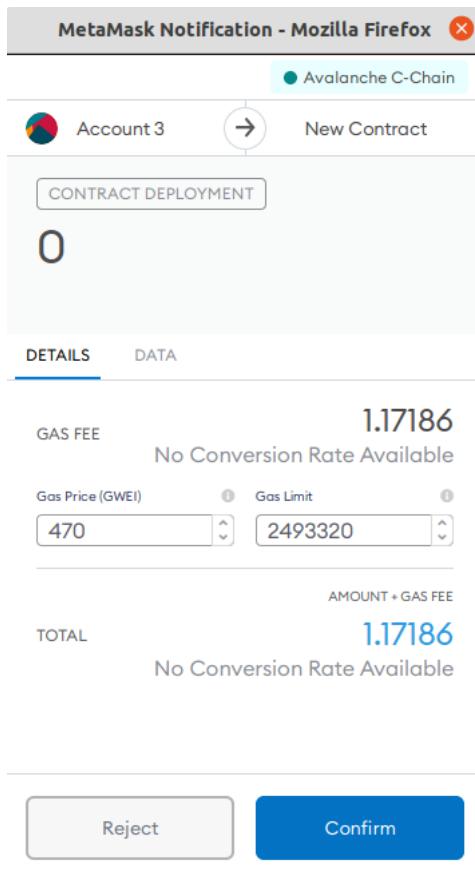
The last thing before the deployment process is to set the contract that will be deployed as a token. Above the Deploy Button, there is a drop-down menu to select a contract. Select the contract named `ERC20PresetMinterPauser.sol`.



Now, here enter the name and symbol of your token. I will name it "test" and the symbol will be `tst`. You can give it a and click to transact button.



After clicking the button, a pop-up will show up and just confirm it.



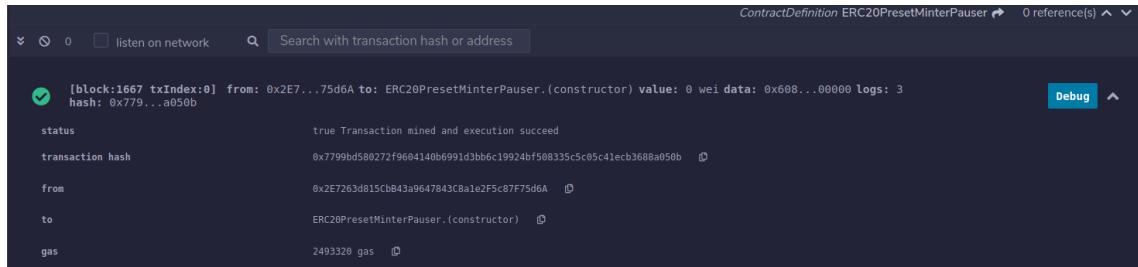
And then another pop-up, a MetaMask confirmation, appears. Confirm it.

After confirming all these pop-ups we have deployed our token to avalanche C-Chain. So we can start to interact with it.

## Interact with Token

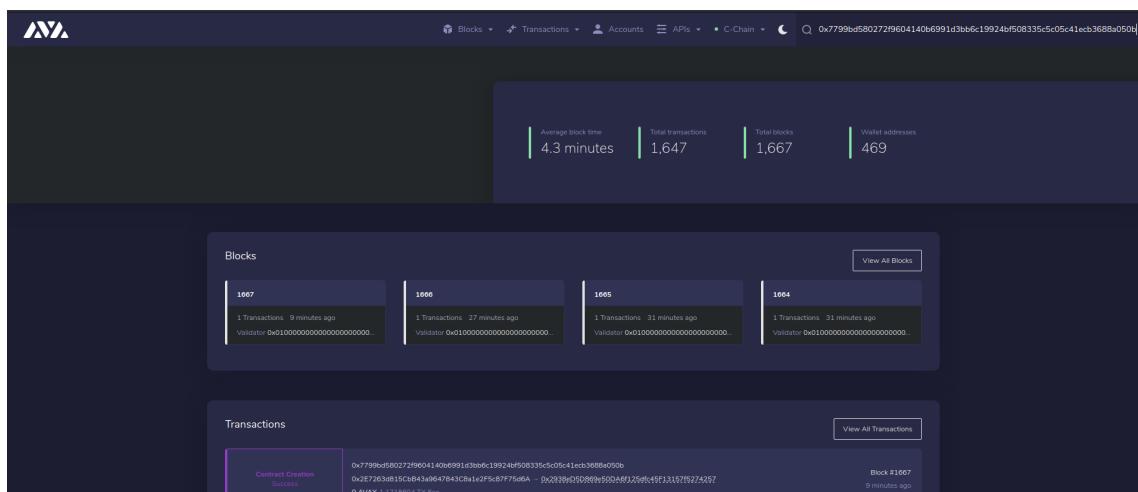
We can see our transaction that deployed on avalanche C-Chain via this [c-chain explorer](#).

But firstly, let's see our transaction hash from the remix console.



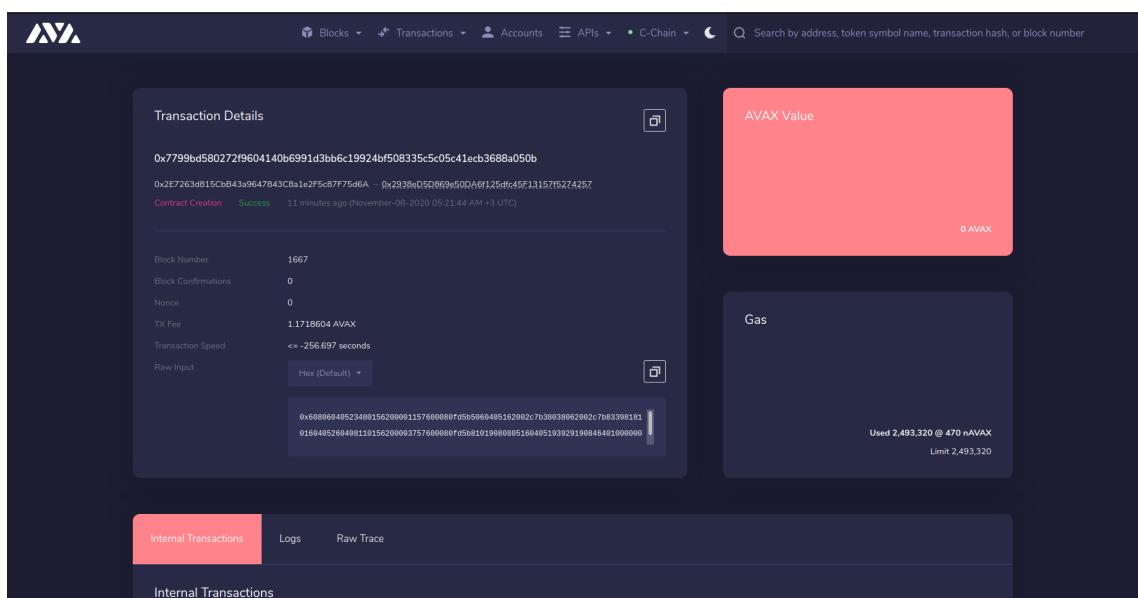
The screenshot shows the Remix IDE's console tab. It displays a green checkmark icon followed by a log entry: "[block:1667 txIndex:0] from: 0x2E7...75d6A to: ERC20PresetMinterPauser.(constructor) value: 0 wei data: 0x608...00000 logs: 3". Below this, detailed transaction parameters are listed: status (true Transaction mined and execution succeed), transaction hash (0x7799bd580272f9604140b6991d3bb6c19924bf508335c5c05c41ecb3688a050b), from (0x2E7263d815cb843a9647843c8a1e2f5c87f75d6A), to (ERC20PresetMinterPauser.(constructor)), and gas (2493320 gas). A "Debug" button is visible in the top right corner.

After deploying the contract, we should see a log in remix console. When you click to arrow and expand it, a transaction hash will come up. Copy it.



The screenshot shows the Avalanche C-Chain Explorer interface. At the top, there are navigation tabs: Blocks, Transactions, Accounts, APIs, C-Chain, and a search bar containing the transaction hash 0x7799bd580272f9604140b6991d3bb6c19924bf508335c5c05c41ecb3688a050b. Below the header, a summary box displays: Average block time (4.3 minutes), Total transactions (1,647), Total blocks (1,667), and Wallet addresses (469). The main content area is divided into two sections: "Blocks" and "Transactions". The "Blocks" section shows four recent blocks (1667, 1666, 1665, 1664) with their respective transaction counts and validators. The "Transactions" section shows a single entry for "Contract Creation Success" with the transaction hash 0x7799bd580272f9604140b6991d3bb6c19924bf508335c5c05c41ecb3688a050b, timestamped at 11 minutes ago (November 08, 2020, 05:21:44 AM +3 UTC). A "View All Transactions" button is located next to this entry.

Just paste the transaction hash to the [explorer](#) I shared above and press enter.



The screenshot shows the Avalanche C-Chain Explorer interface with the transaction hash 0x7799bd580272f9604140b6991d3bb6c19924bf508335c5c05c41ecb3688a050b pasted into the search bar. The results page is titled "Transaction Details". It shows the transaction hash again, the status "Contract Creation Success", and the timestamp "11 minutes ago". Below this, detailed transaction parameters are listed: Block Number (1667), Block Confirmations (0), Nonce (0), TX Fee (1.1718604 AVAX), Transaction Speed (<= 256.697 seconds), and Raw Input (Hex [Default]). To the right, there are three cards: "AVAX Value" (0 AVAX), "Gas" (Used 2,493,320 @ 470 nAVAX, Limit 2,493,320), and "Internal Transactions" (which is currently selected and highlighted in red). Below the "Internal Transactions" card, there are buttons for "Logs" and "Raw Trace".

Here we can see all details about the transaction and token contract.

0xE7263d815CbB43a9647843C8a1e2F5c87F75d6A → 0x2938eD5D869e50DA6f125dfc45F13157f5274257

The first one is my wallet address that creates token and the second address is my token contract address which is named `test`. Now, let's mint some token to our own address.

Transactions recorded 1

Deployed Contracts

ERC20PRESETMINTERPAUSER AT 0X293...74257 (BLOCKCHAIN)

- approve address spender, uint256 amount
- burn uint256 amount
- burnFrom address account, uint256 amount
- decreaseAllowance address spender, uint256 subtractedValue
- grantRole bytes32 role, address account
- increaseAllowance address spender, uint256 addedValue
- mint address to, uint256 amount
- pause
- renounceRole bytes32 role, address account
- revokeRole bytes32 role, address account
- transfer address recipient, uint256 amount

Come back to the remix and after deploying, you should be able to see the contract in "Deployed Contracts" section.

Here, we have a bunch of functions that we can use to interact with our token contract. You can check all these methods from OpenZeppelin documentation to learn how to use them. But we will only use the mint method.

Click to arrow beside the mint method to read it.

mint

to: 0xE7263d815CbB43a9647843C8a1e2F5c87F75d6A

amount: 10000000000000000000000000000000

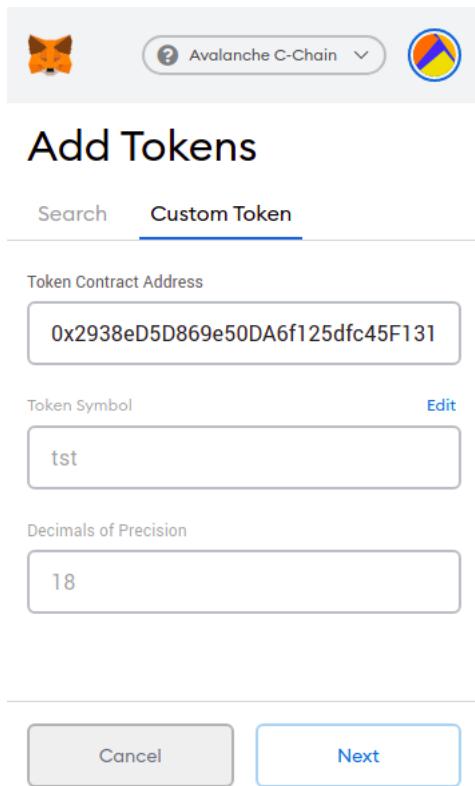
transact

Enter your address and an amount in wei. For example, I will mint 1000 `tst` token so, I entered "1000000000000000000000000"

## Add Token to MetaMask

Now we minted 1000 token to our contract, but you should not be able to see the tokens in your MetaMask wallet. In order to see our own token, we have to add it. On MetaMask, click to "Add Token" button and select "Custom Token" tab.

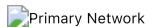
Here enter the token address that you can see from explorer as I showed above. Copy and paste it here. Then click on the Next button, you should see 1000 token that you named in your MetaMask wallet. Also, you can send it to another account via either remix or MetaMask.



**description:** In this doc, learn how to deploy and test a smart contract on Avalanche using Remix and MetaMask.

## Deploy a Smart Contract on Avalanche Using Remix and MetaMask

### Introduction



Avalanche's Primary Network is a Subnet that has three chains: P-Chain, X-Chain, and C-Chain. The C-Chain is an instance of the Ethereum Virtual Machine powered by Avalanche's Snowman consensus protocol. The [C-Chain RPC](#) can do anything a typical Ethereum client can by using the Ethereum-standard RPC calls. The immediate benefits of using the C-Chain rather than Ethereum are all of the benefits of using Avalanche. These properties that could considerably improve the performance of DApps and the user experience.

Today, we will deploy and test a smart contract on Avalanche using Remix and MetaMask.

### Step 1: Setting up MetaMask

Log in to MetaMask -> Click the Network drop-down -> Select Custom RPC



#### Avalanche Mainnet Settings

- **Network Name:** Avalanche Mainnet C-Chain
- **New RPC URL:** <https://api.avax.network/ext/bc/C/rpc>
- **ChainId:** 43114

- **Symbol:** AVAX
- **Explorer:** <https://snowtrace.io/>

#### Fuji Testnet Settings:

- **Network Name:** Avalanche Fuji C-Chain
- **New RPC URL:** <https://api.avax-test.network/ext/bc/C/rpc>
- **ChainID:** 43113
- **Symbol:** AVAX
- **Explorer:** <https://testnet.snowtrace.io/>

#### Local Testnet (Avalanche Network Runner) Settings: ([Avalanche Network Runner Tutorial](#))

- **Network Name:** Avalanche Local C-Chain
- **New RPC URL:** <http://127.0.0.1:34890/ext/bc/C/rpc> (Note: the port number should match your local setting which can be different from 34890.)
- **ChainID:** 43112
- **Symbol:** AVAX
- **Explorer:** N/A

## Step 2: Funding Your C-Chain Address

### Using Avalanche Wallet

On the main net, you can use the [Avalanche Wallet](#) to transfer funds from the X-Chain to your C-Chain address. The process is simple, as explained in this [tutorial](#). Wallet can be used on test and local networks, too.

### Using Test Network Faucet

For funding on the test network, you can use the Test Network Faucet. Navigate to <https://faucet.avax.network/> and paste your C-Chain address. Faucet will automatically know that it needs to send the test AVAX to C-Chain. Click the CAPTCHA checkbox and select 'Request AVAX' button. Your address will receive test AVAX in a few seconds.

### Funding on Local Testnet

On a local network, you can easily fund your addresses by following [this](#).

## Step 3: Connect MetaMask and Deploy a Smart Contract Using Remix

Open [Remix](#) -> Select Solidity

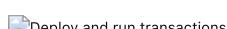


Load or create the smart contracts that we want to compile and deploy using Remix file explorer.

For this example, we will deploy an ERC20 contract from [OpenZeppelin](#).



Navigate to Deploy Tab -> Open the "ENVIRONMENT" drop-down and select Injected Web3 (make sure MetaMask is loaded)



Once we injected the web3-> Go back to the compiler, and compile the selected contract -> Navigate to Deploy Tab



Now, the smart contract is compiled, MetaMask is injected, and we are ready to deploy our ERC20. Click "Deploy."



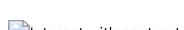
Confirm the transaction on the MetaMask pop up.



Our contract is successfully deployed!



Now, we can expand it by selecting it from the "Deployed Contracts" tab and test it out.



The contract ABI and Bytecode are available on the compiler tab.



If you had any difficulties following this tutorial or simply want to discuss Avalanche with us, you can join our community at [Discord!](#)

## Verify Smart Contracts on the C-Chain Explorer

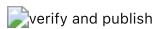
The C-Chain Explorer supports verifying smart contracts, allowing users to review it.

The Mainnet C-Chain Explorer is [here](#) and the Fuji Testnet Explorer is [here](#).

If you have issues, contact us on [Discord](#).

### Steps

Navigate to the Contract tab at the Explorer page for your contract's address.



Click *Verify & Publish* to enter the smart contract verification page.



[Libraries](#) can be provided. If they are, they must be deployed, independently verified and in the *Add Contract Libraries* section.



The C-Chain Explorer can fetch constructor arguments automatically for simple smart contracts. More complicated contracts might require you to pass in special constructor arguments. Smart contracts with complicated constructors [may have validation issues](#). You can try this [online ABI encoder](#).

### Requirements

- **IMPORTANT** Contracts should be verified on Testnet before being deployed to Mainnet to ensure there are no issues.
- Contracts must be flattened.
  - Includes will not work.
- Contracts should be compile-able in [Remix](#).
  - A flattened contract with `pragma experimental ABIEncoderV2` (as an example) can create unusual binary and/or constructor blobs. This might cause validation issues.
- The C-Chain Explorer **only** validates [solc JavaScript](#) and only supports [Solidity](#) contracts.

### Libraries

The compile bytecode will identify if there are external libraries. If you released with Remix, you will also see multiple transactions created.

```
{
 "linkReferences": {
 "contracts/Storage.sol": {
 "MathUtils": [
 {
 "length": 20,
 "start": 3203
 }
 ...
]
 },
 "object": "....",
 ...
}
```

This requires you to add external libraries in order to verify the code.

A library can have dependent libraries. To verify a library, the hierarchy of dependencies will need to be provided to the C-Chain Explorer. Verification may fail if you provide more than the library plus any dependencies (that is you might need to prune the Solidity code to exclude anything but the necessary classes).

You can also see references in the byte code in the form `__$75f20d36....$__`. The keccak256 hash is generated from the library name.

Example [online converter](#): `contracts/Storage.sol:MathUtils => 75f20d361629befd780a5bd3159f017ee0f8283bdb6da80805f83e829337fd12`

### Examples

- [SwapFlashLoan](#)

SwapFlashLoan uses swaputils and mathutils:

- [SwapUtils](#)

SwapUtils requires mathutils:

- [MathUtils](#)

## Caveats

### SPDX License Required

An SPDX must be provided.

```
// SPDX-License-Identifier: ...
```

### keccak256 Strings Processed

The C-Chain Explorer interprets all keccak256(...) strings, even those in comments. This can cause issues with constructor arguments.

```
/// keccak256("1");
keccak256("2")
```

This could cause automatic constructor verification failures. If you receive errors about constructor arguments they can be provided in ABI hex encoded form on the contract verification page.

### Solidity Constructors

Constructors and inherited constructors can cause problems verifying the constructor arguments.

example:

```
abstract contract Parent {
 constructor () {
 address msgSender = ...;
 emit Something(address(0), msgSender);
 }
}
contract Main is Parent {
 constructor (
 string memory _name,
 address deposit,
 uint fee
) {
 ...
 }
}
```

If you receive errors about constructor arguments they can be provided in ABI hex encoded form on the contract verification page.

## Disclaimer

**The Knowledge Base, including all the Help articles on this site, is provided for technical support purposes only, without representation, warranty or guarantee of any kind. Not an offer to sell or solicitation of an offer to buy any security or other regulated financial instrument. Not technical, investment, financial, accounting, tax, legal or other advice; please consult your own professionals. Please conduct your own research before connecting to or interacting with any dapp or third party or making any investment or financial decisions. MoonPay, ParaSwap and any other third party services or dapps you access are offered by third parties unaffiliated with us. Please review this [Notice](#) and the [Terms of Use](#).**

**description: Avalanche is a consensus protocol that is scalable, robust, and decentralized.**

## Avalanche Consensus

Consensus is the task of getting a group of computers (a.k.a. nodes) to come to an agreement on a decision. In blockchain, this means that all the participants in a network have to agree on the changes made to the shared ledger. This agreement is reached through a specific process, a consensus protocol, that ensures that everyone sees the same information and that the information is accurate and trustworthy.

**Avalanche Consensus** is a consensus protocol that is scalable, robust, and decentralized. It combines features of both classical and Nakamoto consensus mechanisms to achieve high throughput, fast finality, and energy efficiency. For the whitepaper, see [here](#).

Key Features Include:

- Speed: Avalanche consensus provides sub-second, immutable finality, ensuring that transactions are quickly confirmed and irreversible.
- Scalability: Avalanche consensus enables high network throughput while ensuring low latency.
- Energy Efficiency: Unlike other popular consensus protocols, participation in Avalanche consensus is neither computationally intensive nor expensive.
- Adaptive Security: Avalanche consensus is designed to resist various attacks, including sybil attacks, distributed denial-of-service (DDoS) attacks, and collusion attacks. Its probabilistic nature ensures that the consensus outcome converges to the desired state, even when the network is under attack.



## Conceptual Overview

Consensus protocols in the Avalanche family operate through repeated sub-sampled voting. When a node is determining whether a [transaction](#) should be accepted, it asks a small, random subset of [validator nodes](#) for their preference. Each queried validator replies with the transaction that it prefers, or thinks should be accepted.

:::note

Consensus will never include a transaction that is determined to be **invalid**. For example, if you were to submit a transaction to send 100 AVAX to a friend, but your wallet only has 2 AVAX, this transaction is considered **invalid** and will not participate in consensus.

:::

If a sufficient majority of the validators sampled reply with the same preferred transaction, this becomes the preferred choice of the validator that inquired.

In the future, this node will reply with the transaction preferred by the majority.

The node repeats this sampling process until the validators queried reply with the same answer for a sufficient number of consecutive rounds.

- The number of validators required to be considered a "sufficient majority" is referred to as " $\alpha$ " (*alpha*).
- The number of consecutive rounds required to reach consensus, a.k.a. the "Confidence Threshold," is referred to as " $\beta$ " (*beta*).
- Both  $\alpha$  and  $\beta$  are configurable.

When a transaction has no conflicts, finalization happens very quickly. When conflicts exist, honest validators quickly cluster around conflicting transactions, entering a positive feedback loop until all correct validators prefer that transaction. This leads to the acceptance of non-conflicting transactions and the rejection of conflicting transactions.



Avalanche Consensus guarantees that if any honest validator accepts a transaction, all honest validators will come to the same conclusion.

:::info

For a great visualization, check out [this demo](#).

:::

## Deep Dive Into Avalanche Consensus



### Intuition

First, let's develop some intuition about the protocol. Imagine a room full of people trying to agree on what to get for lunch. Suppose it's a binary choice between pizza and barbecue. Some people might initially prefer pizza while others initially prefer barbecue. Ultimately, though, everyone's goal is to achieve **consensus**.

Everyone asks a random subset of the people in the room what their lunch preference is. If more than half say pizza, the person thinks, "OK, looks like things are leaning toward pizza. I prefer pizza now." That is, they adopt the *preference* of the majority. Similarly, if a majority say barbecue, the person adopts barbecue as their preference.

Everyone repeats this process. Each round, more and more people have the same preference. This is because the more people that prefer an option, the more likely someone is to receive a majority reply and adopt that option as their preference. After enough rounds, they reach consensus and decide on one option,

which everyone prefers.

### Snowball

The intuition above outlines the Snowball Algorithm, which is a building block of Avalanche consensus. Let's review the Snowball algorithm.

#### Parameters

- $n$ : number of participants
- $k$  (sample size): between 1 and  $n$
- $\alpha$  (quorum size): between 1 and  $k$
- $\beta$  (decision threshold):  $\geq 1$

#### Algorithm

```
preference := pizza
consecutiveSuccesses := 0
while not decided:
 ask k random people their preference
 if >= α give the same response:
 preference := response with >= α
 if preference == old preference:
 consecutiveSuccesses++
 else:
 consecutiveSuccesses = 1
 else:
 consecutiveSuccesses = 0
 if consecutiveSuccesses > β:
 decide(preference)
```

#### Algorithm Explained

Everyone has an initial preference for pizza or barbecue. Until someone has *decided*, they query  $k$  people (the sample size) and ask them what they prefer. If  $\alpha$  or more people give the same response, that response is adopted as the new preference.  $\alpha$  is called the *quorum size*. If the new preference is the same as the old preference, the `consecutiveSuccesses` counter is incremented. If the new preference is different than the old preference, the `consecutiveSuccesses` counter is set to `1`. If no response gets a quorum (an  $\alpha$  majority of the same response) then the `consecutiveSuccesses` counter is set to `0`.

Everyone repeats this until they get a quorum for the same response  $\beta$  times in a row. If one person decides pizza, then every other person following the protocol will eventually also decide on pizza.

Random changes in preference, caused by random sampling, cause a network preference for one choice, which begets more network preference for that choice until it becomes irreversible and then the nodes can decide.

In our example, there is a binary choice between pizza or barbecue, but Snowball can be adapted to achieve consensus on decisions with many possible choices.

The liveness and safety thresholds are parameterizable. As the quorum size,  $\alpha$ , increases, the safety threshold increases, and the liveness threshold decreases. This means the network can tolerate more byzantine (deliberately incorrect, malicious) nodes and remain safe, meaning all nodes will eventually agree whether something is accepted or rejected. The liveness threshold is the number of malicious participants that can be tolerated before the protocol is unable to make progress.

These values, which are constants, are quite small on the Avalanche Network. The sample size,  $k$ , is `20`. So when a node asks a group of nodes their opinion, it only queries `20` nodes out of the whole network. The quorum size,  $\alpha$ , is `14`. So if `14` or more nodes give the same response, that response is adopted as the querying node's preference. The decision threshold,  $\beta$ , is `20`. A node decides on choice after receiving `20` consecutive quorum ( $\alpha$  majority) responses.

Snowball is very scalable as the number of nodes on the network,  $n$ , increases. Regardless of the number of participants in the network, the number of consensus messages sent remains the same because in a given query, a node only queries `20` nodes, even if there are thousands of nodes in the network.

Everything discussed to this point is how Avalanche is described in [the Avalanche white-paper](#). The implementation of the Avalanche consensus protocol by Ava Labs (namely in AvalancheGo) has some optimizations for latency and throughput.

### Blocks

A block is a fundamental component that forms the structure of a blockchain. It serves as a container or data structure that holds a collection of transactions or other relevant information. Each block is cryptographically linked to the previous block, creating a chain of blocks, hence the term "blockchain."

In addition to storing a reference of its parent, a block contains a set of transactions. These transactions can represent various types of information, such as financial transactions, smart contract operations, or data storage requests.

If a node receives a vote for a block, it also counts as a vote for all of the block's ancestors (its parent, the parents' parent, etc.).

### Finality

Avalanche consensus is probabilistically safe up to a safety threshold. That is, the probability that a correct node accepts a transaction that another correct node rejects can be made arbitrarily low by adjusting system parameters. In Nakamoto consensus protocol (as used in Bitcoin and Ethereum, for example), a block may be included in the chain but then be removed and not end up in the canonical chain. This means waiting an hour for transaction settlement. In Avalanche, acceptance/rejection are **final and irreversible** and only take a few seconds.

### Optimizations

It's not safe for nodes to just ask, "Do you prefer this block?" when they query validators. In Ava Labs' implementation, during a query a node asks, "Given that this block exists, which block do you prefer?" Instead of getting back a binary yes/no, the node receives the other node's preferred block.

Nodes don't only query upon hearing of a new block; they repeatedly query other nodes until there are no blocks processing.

Nodes may not need to wait until they get all  $k$  query responses before registering the outcome of a poll. If a block has already received  $\alpha$  votes, then there's no need to wait for the rest of the responses.

### Validators

If it were free to become a validator on the Avalanche network, that would be problematic because a malicious actor could start many, many nodes which would get queried very frequently. The malicious actor could make the node act badly and cause a safety or liveness failure. The validators, the nodes which are queried as part of consensus, have influence over the network. They have to pay for that influence with real-world value in order to prevent this kind of ballot stuffing. This idea of using real-world value to buy influence over the network is called Proof of Stake.

To become a validator, a node must **bond** (stake) something valuable (**AVAX**). The more AVAX a node bonds, the more often that node is queried by other nodes. When a node samples the network it's not uniformly random. Rather, it's weighted by stake amount. Nodes are incentivized to be validators because they get a reward if, while they validate, they're sufficiently correct and responsive.

Avalanche doesn't have slashing. If a node doesn't behave well while validating, such as giving incorrect responses or perhaps not responding at all, its stake is still returned in whole, but with no reward. As long as a sufficient portion of the bonded AVAX is held by correct nodes, then the network is safe, and is live for virtuous transactions.

### Big Ideas

Two big ideas in Avalanche are **subsampling** and **transitive voting**.

Subsampling has low message overhead. It doesn't matter if there are twenty validators or two thousand validators; the number of consensus messages a node sends during a query remains constant.

Transitive voting, where a vote for a block is a vote for all its ancestors, helps with transaction throughput. Each vote is actually many votes in one.

### Loose Ends

Transactions are created by users which call an API on an [AvalancheGo](#) full node or create them using a library such as [AvalancheJS](#).

### Other Observations

Conflicting transactions are not guaranteed to be live. That's not really a problem because if you want your transaction to be live then you should not issue a conflicting transaction.

Snowman is the name of Ava Labs' implementation of the Avalanche consensus protocol for linear chains.

If there are no undecided transactions, the Avalanche consensus protocol *quiesce*. That is, it does nothing if there is no work to be done. This makes Avalanche more sustainable than Proof-of-work where nodes need to constantly do work.

Avalanche has no leader. Any node can propose a transaction and any node that has staked AVAX can vote on every transaction, which makes the network more robust and decentralized.

## Why Do We Care?

Avalanche is a general consensus engine. It doesn't matter what type of application is put on top of it. The protocol allows the decoupling of the application layer from the consensus layer. If you're building a dapp on Avalanche then you just need to define a few things, like how conflicts are defined and what is in a transaction. You don't need to worry about how nodes come to an agreement. The consensus protocol is a black box that puts something into it and it comes back as accepted or rejected.

Avalanche can be used for all kinds of applications, not just P2P payment networks. Avalanche's Primary Network has an instance of the Ethereum Virtual Machine, which is backward compatible with existing Ethereum Dapps and dev tooling. The Ethereum consensus protocol has been replaced with Avalanche consensus to enable lower block latency and higher throughput.

Avalanche is very performant. It can process thousands of transactions per second with one to two second acceptance latency.

## Summary

**Avalanche consensus is a radical breakthrough in distributed systems. It represents as large a leap forward as the classical and Nakamoto consensus protocols that came before it. Now that you have a better understanding of how it works, check out other documentations for building game-changing Dapps and financial instruments on Avalanche.**

**description:** Avalanche features 3 built-in blockchains which includes Exchange Chain (X-Chain), Platform Chain (P-Chain), and Contract Chain (C-Chain). More info here. **sidebar\_label:** The Primary Network

## The Primary Network

Avalanche is a heterogeneous network of blockchains. As opposed to homogeneous networks, where all applications reside in the same chain, heterogeneous networks allow separate chains to be created for different applications.

The Primary Network is a special [Subnet](#) that runs three blockchains:

- The Contract Chain ([C-Chain](#))
- The Platform Chain ([P-Chain](#))
- The Exchange Chain ([X-Chain](#))

:::note Avalanche Mainnet is comprised of the Primary Network and all deployed Subnets. :::

A node can become a validator for the Primary Network by staking at least **2,000 AVAX**.



## The Chains

All validators of the Primary Network are required to validate and secure the following:

### C-Chain

The **C-Chain** is an implementation of the Ethereum Virtual Machine (EVM). The [C-Chain's API](#) supports Geth's API and supports the deployment and execution of smart contracts written in Solidity.

The C-Chain is an instance of the [Coreth](#) Virtual Machine.

### P-Chain

The **P-Chain** is responsible for all validator and Subnet-level operations. The [P-Chain API](#) supports the creation of new blockchains and Subnets, the addition of validators to Subnets, staking operations, and other platform-level operations.

The P-Chain is an instance of the Platform Virtual Machine.

### X-Chain

The **X-Chain** is responsible for operations on digital smart assets known as **Avalanche Native Tokens**. A smart asset is a representation of a real-world resource (for example, equity, or a bond) with sets of rules that govern its behavior, like "can't be traded until tomorrow." The [X-Chain API](#) supports the creation and trade of Avalanche Native Tokens.

One asset traded on the X-Chain is AVAX. When you issue a transaction to a blockchain on Avalanche, you pay a fee denominated in AVAX.

## The X-Chain is an instance of the Avalanche Virtual Machine (AVM).

### sidebar\_label: AVAX Token

## AVAX

AVAX is the native utility token of Avalanche. It's a hard-capped, scarce asset that is used to pay for fees, secure the platform through staking, and provide a basic unit of account between the multiple Subnets created on Avalanche.

:::info

- 1 nAVAX is equal to 0.000000001 AVAX .

:::

## Utility

AVAX is a capped-supply (up to 720M) resource in the Avalanche ecosystem that's used to power the network. AVAX is used to secure the ecosystem through staking and for day-to-day operations like issuing transactions.

AVAX represents the weight that each node has in network decisions. No single actor owns the Avalanche Network, so each validator in the network is given a proportional weight in the network's decisions corresponding to the proportion of total stake that they own through proof of stake (PoS).

Any entity trying to execute a transaction on Avalanche pays a corresponding fee (commonly known as "gas") to run it on the network. The fees used to execute a transaction on Avalanche is burned, or permanently removed from circulating supply.

## Tokenomics

A fixed amount of 360M AVAX was minted at genesis, but a small amount of AVAX is constantly minted as a reward to validators. The protocol rewards validators for good behavior by minting them AVAX rewards at the end of their staking period. The minting process offsets the AVAX burned by transactions fees. While AVAX is still far away from its supply cap, it will almost always remain an inflationary asset.

Avalanche does not take away any portion of a validator's already staked tokens (commonly known as "slashing") for negligent/malicious staking periods, however this behavior is disincentivized as validators who attempt to do harm to the network would expend their node's computing resources for no reward.

AVAX is minted according to the following formula, where  $R_j$  is the total number of tokens at year  $j$ , with  $R_1 = 360M$ , and  $R_{j-1}$  representing the last year that the values of  $\gamma, \lambda$  were changed;  $c_j$  is the yet un-minted supply of coins to reach  $720M$  at year  $j$  such that  $c_j \leq 360M$ ;  $u$  represents a staker, with  $u.s_{amount}$  representing the total amount of stake that  $u$  possesses, and  $u.s_{time}$  the length of staking for  $u$ .

AVAX is minted according to the following formula, where  $R_j$  is the total number of tokens at:

$$R_j = R_{j-1} + \sum_{u \in \text{validators}} \rho(u.s_{amount}, u.s_{time}) \times \left( \frac{c_{j-1}}{L} \right) \times \left( \frac{\sum_{i=0}^{j-1} \left( \frac{1}{\gamma} + \frac{\lambda}{\gamma} \right)^i}{\left( \frac{1}{\gamma} - 1 \right)} \right)$$

where

$$L = \left( \sum_{i=0}^{\infty} \left( \frac{1}{\gamma} + \frac{\lambda}{\gamma} \right)^i \right)$$

At genesis,  $c_1 = 360M$ . The values of  $\gamma$  and  $\lambda$  are governable, and if changed, the function is recomputed with the new value of  $c_*$ . We have that  $\sum \rho(u.s_{amount}, u.s_{time})$  is a linear function that can be computed as follows ( $u.s_{time}$  is measured in weeks, and  $u.s_{amount}$  is measured in AVAX tokens):

$\sum_{i=1}^n \rho(u.s_{-}(amount), u.s_{-}(time)) = (0.002 \times u.s_{-}(time) + 0.896) \times \frac{u.s_{-}(amount)}{R_i}$

If the entire supply of tokens at year  $t$  is staked for the maximum amount of staking time (one year, or 52 weeks), then  $\sum_{i=1}^n \rho(u.s_{-}(amount), u.s_{-}(time)) = 1$ . If, instead, every token is staked continuously for the minimal stake duration of two weeks, then  $\sum_{i=1}^n \rho(u.s_{-}(amount), u.s_{-}(time)) = 0.9$ . Therefore, staking for the maximum amount of time incurs an additional 11.11% of tokens minted, incentivizing stakers to stake for longer periods.

Due to the capped-supply, the above function guarantees that regardless of the governance changes, AVAX will never exceed a total of \$720M tokens, or  $\lim_{t \rightarrow \infty} R(t) = 720M$ .

---

**description:** Fuji testnet is the official Avalanche testnet. **sidebar\_label:** Fuji Testnet

## Fuji Testnet

The Fuji Testnet serves as the official testnet for the Avalanche ecosystem.

Fuji's infrastructure imitates Avalanche Mainnet. It's comprised of a [Primary Network](#) formed by instances of X, P, and C-Chain, as well as many test Subnets.

### Why Use Fuji Over Mainnet?

Fuji provides users with a platform to simulate the conditions found in the Mainnet environment. It enables developers to deploy demo Smart Contracts, allowing them to test and refine their applications before deploying them on the [Primary Network](#).

Users interested in experimenting with Avalanche can receive free testnet AVAX, allowing them to explore the platform without any risk. These testnet tokens have no value in the real world and are only meant for experimentation purposes within the Fuji test network.

To receive testnet tokens, users can request funds from the [Avalanche Faucet](#).

### Additional Considerations

- Fuji Testnet has its own dedicated block explorer on [Snowtrace](#).
- The Public API endpoint for Fuji is not the same as Mainnet. More info is available in the [Public API Server](#) documentation.
- While Fuji Network is a valuable resource, developers also have the option to explore [Avalanche Network Runner](#) as an alternative means of locally testing their projects, ensuring comprehensive evaluation and fine-tuning before interacting with the wider network.

---

**slug:** /intro **sidebar\_label:** ▲ What is Avalanche?

## What Is Avalanche?

Avalanche is an open-source platform for building decentralized applications in one interoperable, decentralized, and highly scalable ecosystem. Powered by a [unique consensus mechanism](#), Avalanche is the first ecosystem designed to accommodate the scale of global finance, with near-instant transaction finality.

### Why Choose Avalanche?

#### Blazingly Fast

- Avalanche employs the fastest consensus mechanism of any layer 1 blockchain. The unique consensus mechanism enables quick finality and low latency: in less than 2 seconds, your transaction is effectively processed and verified.

#### Built to Scale

- Developers who build on Avalanche can build application-specific blockchains with complex rulesets or build on existing private or public Subnets in any language.
- Avalanche is incredibly energy-efficient and can run easily on consumer-grade hardware. The entire Avalanche network consumes the same amount of energy as 46 US households, equivalent to 0.0005% of the amount of energy consumed by Bitcoin.
- Solidity developers can build on Avalanche's implementation of the EVM straight out-of-the box, or build their own custom Virtual Machine (VM) for advanced use cases.

#### Advanced Security

- Avalanche consensus scales to thousands of concurrent validators without suffering performance degradation making it one of the most secure protocols for internet scaling systems.
- Permissionless and permissioned custom blockchains deployed as an Avalanche Subnets can include custom rulesets designed to be compliant with legal and jurisdictional considerations.

---

**sidebar\_label:** Subnets

## What Is a Subnet?

A **Subnet** is a sovereign network which defines its own rules regarding its membership and token economics. It is composed of a dynamic subset of Avalanche validators working together to achieve consensus on the state of one or more blockchains. Each blockchain is validated by exactly one Subnet, while a Subnet can validate many blockchains.

Avalanche's [Primary Network](#) is a special Subnet running three blockchains:

- The Platform Chain ([P-Chain](#))
- The Contract Chain ([C-Chain](#))

- The Exchange Chain ([X-Chain](#))



(Image adopted from [this article](#))

:::info Every validator in a Subnet **must** also validate the Primary Network. :::

Node operators that validate a Subnet with multiple chains do not need to run multiple machines for validation. For example, the Primary Network is a Subnet with three coexisting chains, all of which can be validated by a single node, or a single machine.

## Advantages

### Independent Networks

- Subnets use virtual machines to specify their own execution logic, determine their own fee regime, maintain their own state, facilitate their own networking, and provide their own security.
- Each Subnet's performance is isolated from other Subnets in the ecosystem, so increased usage on one Subnet won't affect another.
- Subnets can have their own token economics with their own native tokens, fee markets, and incentives determined by the Subnet deployer.
- One Subnet can host multiple blockchains with customized [virtual machines](#).

### Native Interoperability

- Avalanche Warp Messaging enables native cross-Subnet communication and allows Virtual Machine (VM) developers to implement arbitrary communication protocols between any two Subnets.

### Accommodate Application-Specific Requirements

*Different blockchain-based applications may require validators to have certain properties such as large amounts of RAM or CPU power.*

- A Subnet could require that validators meet certain [hardware requirements](#) so that the application doesn't suffer from low performance due to slow validators.

### Launch a Network Designed With Compliance In Mind

*Avalanche's Subnet architecture makes regulatory compliance manageable. As mentioned above, a Subnet may require validators to meet a set of requirements.*

Some examples of requirements the creators of a Subnet may choose include:

- Validators must be located in a given country.
- Validators must pass KYC/AML checks.
- Validators must hold a certain license.

### Control The Privacy of On-Chain Data

*Subnets are ideal for organizations interested in keeping their information private.*

- Institutions conscious of their stakeholders' privacy can create a private Subnet where the contents of the blockchains would be visible only to a set of pre-approved validators. Define this at creation with a [single parameter](#).

### Validator Sovereignty

*In a heterogeneous network of blockchains, some validators will not want to validate certain blockchains because they simply have no interest in those blockchains.*

- The Subnet model enables validators to concern themselves only with blockchain networks they choose to participate in. This greatly reduces the computational burden on validators.

## Develop Your Own Subnet

To get started, check out the tutorials in our [Subnets](#) section.

## Virtual Machines

In a nutshell, a **Virtual Machine** (VM) is the blueprint for a blockchain, meaning it defines the application-level logic of a blockchain. In technical terms, it specifies the blockchain's state, state transition function, transactions, and the API through which users can interact with the blockchain.

You can use the same VM to create many blockchains, each of which follows the same rule-set but is independent of all others.

### Why Run a VM on Avalanche?

*Developers with advanced use-cases for utilizing distributed ledger technology are often forced to build their own blockchain from scratch, re-implement complex abstractions like networking and consensus, all before even being able to start working on their new application.*

#### With Avalanche,

- Developers can create a VM that defines how their blockchain should behave, and use this blueprint to coordinate validators in the network to run the application.
- VMs can be written in any language, and use libraries and tech stacks that its creator is familiar with. Developers have fine control over the behavior of their blockchain, and can redefine the rules of a blockchain to fit any use-case they have.

- Developers don't need to concern yourself with lower-level logic like networking, consensus, and the structure of the blockchain; Avalanche does this behind the scenes so you can focus on building your Dapp, your ecosystem, and your community.

## How VMs Work

VMs communicate with Avalanche over a language agnostic request-response protocol known as [RPC](#). This allows the VM framework to open a world of endless possibilities, as developers can implement their Dapps using the languages, frameworks, and libraries of their choice. To get started, create a VM out-of-the-box with the [Subnet-EVM](#) in Solidity, or design a custom VM with languages like Golang, Rust, and many more.

## Running a VM

All Avalanche validators as members of the Avalanche Primary Network are required to run three VMs:

- Coreth: Defines the Contract Chain (C-Chain); supports smart contract functionality and is EVM-compatible.
- Platform VM: Defines the Platform Chain (P-Chain); supports operations on staking and Subnets.
- Avalanche VM: Defines the Exchange Chain (X-Chain); supports operations on Avalanche Native Tokens.

All three can easily be run on any computer with [AvalancheGo](#).

Validators can install additional VMs on their node to validate additional [Subnets](#) in the Avalanche ecosystem. In exchange, validators receive staking rewards in the form of a reward token determined by the Subnets.

## Solidity

Avalanche natively supports the execution of Ethereum smart contracts written in solidity. Ethereum developers have the option of deploying their smart contracts on the C-Chain's implementation of the Ethereum Virtual Machine ([Coreth](#)), or on their own Subnet using the [Subnet-EVM](#) for advanced use cases that require more customization.

Both C-Chain and the Subnet-EVM are compatible with Ethereum tooling like Remix, MetaMask, Truffle, and more.

To learn more about smart contract support, click [here](#).

## Golang

- [Coreth](#)
  - An implementation of the EVM that powers the Avalanche C-Chain that supports Solidity smart contracts.
- [Subnet-EVM](#)
  - An implementation of the EVM that can be deployed to a custom Subnet to support Solidity smart contracts
- [TimestampVM](#)
  - A decentralized timestamp server
- [XSVM](#)
  - An example of Avalanche Warp Messaging that implements Cross-Subnet asset transfers

See here for a tutorial on [How to Build a Simple Golang VM](#)

## Rust

The following VMs were built using Rust via the [Avalanche Rust SDK](#)

- [TimestampVM RS](#)
  - A Rust implementation of TimestampVM

See here for a tutorial on [How to Build a Simple Rust VM](#)

## Projects

### AvalancheGo

[Link to GitHub](#)

Go implementation of an Avalanche node.

Related docs

- [AvalancheGo](#)
- [Nodes](#)

### Coreth

[Link to GitHub](#)

Coreth (from core Ethereum) is the [Virtual Machine \(VM\)](#) that defines the Avalanche Contract Chain (C-Chain).

Related docs

- [C-Chain APIs](#)

### Subnet-EVM

[Link to GitHub](#)

Subnet-EVM is the [Virtual Machine \(VM\)](#), that defines the Subnet Contract Chains. Subnet-EVM is a simplified version of [Coreth VM \(C-Chain\)](#).

Related docs

- [Subnets](#)

## Avalanche-CLI

[Link to GitHub](#)

Avalanche CLI is a command line tool that gives developers access to everything Avalanche. This release specializes in helping developers develop and test Subnets.

Related docs

- [Avalanche-CLI](#)

## AvalancheJS

[Link to GitHub](#)

The Avalanche platform JavaScript library.

Related docs

- [AvalancheJS](#)

## Avalanche Wallet

[Link to GitHub](#)

Related docs

- [Wallet](#)

## Avalanche Network Runner

[Link to GitHub](#)

This is a tool to run and interact with a local Avalanche network.

Related docs

- [Avalanche Network Runner](#)

## Avalanche Developer Docs

[Link to GitHub](#)

**Avalanche developer documentation and tutorials.**

**sidebar\_position: 1**

# Nodes Overview

This section provides documents on how to build and maintain an AvalancheGo node, and then validate the Avalanche network using an AvalancheGo node.

## Build

|                                                                      |                                                        |
|----------------------------------------------------------------------|--------------------------------------------------------|
| <a href="#">Run an Avalanche Node Using Install Script</a>           | How to install and run AvalancheGo                     |
| <a href="#">Run an Avalanche Node Manually</a>                       | How to install without the installer script            |
| <a href="#">Run an Avalanche Node with Amazon Web Services (AWS)</a> | Create a node that runs on AWS                         |
| <a href="#">Launch an Avalanche Validator on AWS with One Click</a>  | Deploy a node that runs on AWS via a one-click install |
| <a href="#">Run an Avalanche Node with Microsoft Azure</a>           | Create a node that runs on Microsoft Azure             |
| <a href="#">Run an Avalanche Node with Google Cloud Platform</a>     | Create a node that runs on Google GCP                  |
| <a href="#">Run an Avalanche Node with Alibaba Cloud</a>             | Create a node that runs on Alibaba Cloud               |
| <a href="#">Run an Avalanche Node with Tencent Cloud</a>             | Create a node that runs on Tencent Cloud               |
| <a href="#">Frequently Asked Questions</a>                           | Learn about common errors when building your node.     |

## Maintain

|                                               |                                                             |
|-----------------------------------------------|-------------------------------------------------------------|
| <a href="#">Node Backup and Restore</a>       | Back up important files to be able to restore your node     |
| <a href="#">Upgrade Your AvalancheGo Node</a> | Upgrade your Avalanche node                                 |
| <a href="#">Monitor an Avalanche Node</a>     | Set up infrastructure to monitor an instance of AvalancheGo |

|                                              |                                               |
|----------------------------------------------|-----------------------------------------------|
| <a href="#">AvalancheGo Config and Flags</a> | Configuration and flags for an Avalanche node |
| <a href="#">Chain Configs</a>                | Chain specific configurations                 |
| <a href="#">Subnet Configs</a>               | Subnet specific configurations                |
| <a href="#">Run C-Chain Offline Pruning</a>  | C-Chain offline pruning                       |
| <a href="#">Node Bootstrap</a>               | Understand how a node bootstraps              |

## Validate

|                                  |                                                   |
|----------------------------------|---------------------------------------------------|
| <a href="#">What is Staking?</a> | Explain the concept of staking                    |
| <a href="#">Add a Validator</a>  | Add a node to the <a href="#">Primary Network</a> |

## Frequently Asked Questions

### Introduction

If you experience any issues building your node, here are some common errors and possible solutions.

**Error:** `WARN node/node.go:291 failed to connect to bootstrap nodes`

This error can occur when the node doesn't have access to the Internet or if the NodeID is already being used by a different node in the network. This can occur when an old instance is running and not terminated.

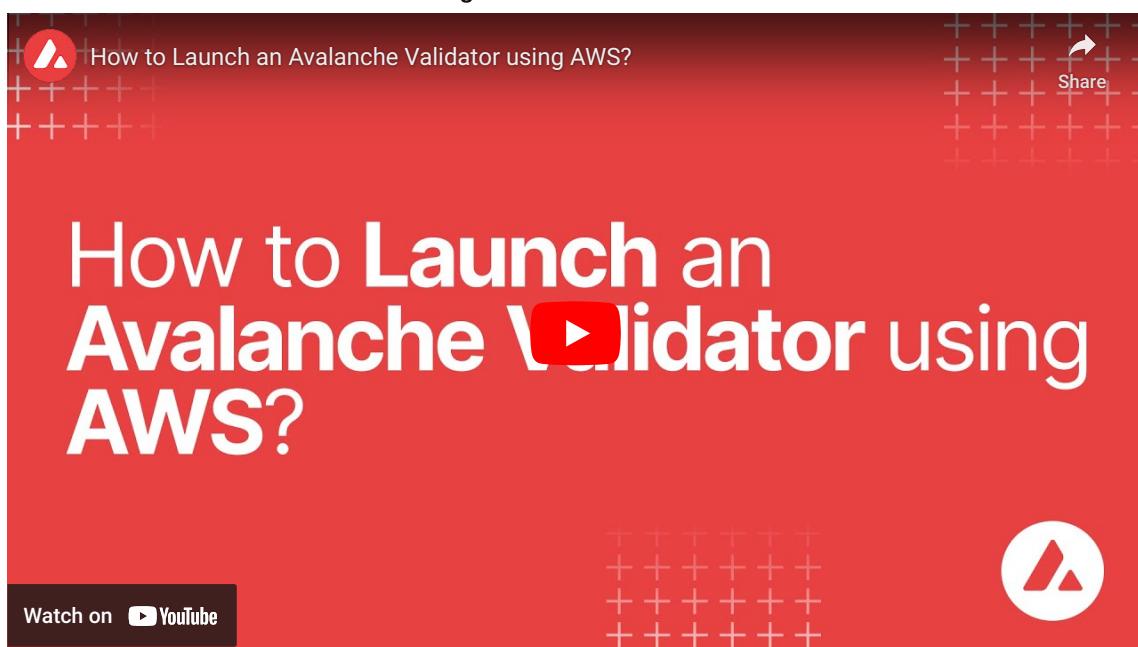
**Error:** `err="cannot query unfinalized data"`

There may be a number of reasons for this issue, but it is likely that the node is not connected properly to other validators, which is usually caused by networking misconfiguration (wrong public IP, closed p2p port 9651).

**sidebar\_position: 4 description:** This tutorial will guide you through spinning up an Avalanche node via the one-click validator node through the AWS Marketplace. This includes subscribing to the software, launching it on EC2, connecting to the node over ssh, calling curl commands, adding the node as a validator on the Fuji network using the Avalanche Web wallet, and confirming the node is a pending validator.

## Launch an Avalanche Validator on AWS with One Click

### How to Launch an Avalanche Validator using AWS



### Introduction

Avalanche is an open-source platform for launching decentralized applications and enterprise blockchain deployments in one interoperable, highly scalable ecosystem. Avalanche is the first decentralized smart contracts platform built for the scale of global finance, with near-instant transaction finality.

With the intention of enabling developers and entrepreneurs to on-ramp into the Avalanche ecosystem with as little friction as possible, Ava Labs recently launched an offering to deploy an Avalanche Validator node via the AWS Marketplace. This tutorial will show the main steps required to get this node running and validating on the Avalanche Fuji testnet.

## Product Overview

The Avalanche Validator node is available via [the AWS Marketplace](#). There you'll find a high level product overview. This includes a product description, pricing information, usage instructions, support information and customer reviews. After reviewing this information you want to click the "Continue to Subscribe" button.

## Subscribe to This Software

Once on the "Subscribe to this Software" page you will see a button which enables you to subscribe to this AWS Marketplace offering. In addition you'll see Terms of service including the seller's End User License Agreement and the [AWS Privacy Notice](#). After reviewing these you want to click on the "Continue to Configuration" button.

## Configure This Software

This page lets you choose a fulfillment option and software version to launch this software. No changes are needed as the default settings are sufficient. Leave the Fulfillment Option as 64-bit (x86) Amazon Machine Image (AMI). The software version is the latest build of [the Avalanche Go full node](#), v1.9.5 (Dec 22, 2022), AKA Banff.5. This will always show the latest version. Also, the Region to deploy in can be left as US East (N. Virginia). On the right you'll see the software and infrastructure pricing. Lastly, click the "Continue to Launch" button.

## Launch This Software

Here you can review the launch configuration details and follow the instructions to launch the Avalanche Validator Node. The changes are very minor. Leave the action as "Launch from Website." The EC2 Instance Type should remain c5.2xlarge. The primary change you'll need to make is to choose a keypair which will enable you to ssh into the newly created EC2 instance to run curl commands on the Validator node. You can search for existing Keypairs or you can create a new keypair and download it to your local machine. If you create a new keypair you'll need to move the keypair to the appropriate location, change the permissions and add it to the OpenSSH authentication agent. For example, on MacOS it would look similar to the following:

```
In this example we have a keypair called avalanche.pem which was downloaded from AWS to ~/Downloads/avalanche.pem
Confirm the file exists with the following command
test -f ~/Downloads/avalanche.pem && echo "avalanche.pem exists."

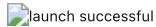
Running the above command will output the following:
avalanche.pem exists.

Move the avalanche.pem keypair from the ~/Downloads directory to the hidden ~/.ssh directory
mv ~/Downloads/avalanche.pem ~/.ssh

Next add the private key identity to the OpenSSH authentication agent
ssh-add ~/.ssh/avalanche.pem;

Change file modes or Access Control Lists
sudo chmod 600 ~/.ssh/avalanche.pem
```

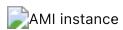
Once these steps are complete you are ready to launch the Validator node on EC2. To make that happen click the "Launch" button



You now have an Avalanche node deployed on an AWS EC2 instance! Copy the AMI ID and click on the EC2 Console link for the next step.

## EC2 Console

Now take the AMI ID from the previous step and input it into the search bar on the EC2 Console. This will bring you to the dashboard where you can find the EC2 instances public IP address.



Copy that public IP address and open a Terminal or command line prompt. Once you have the new Terminal open ssh into the EC2 instance with the following command.

```
ssh username@ip.address.of.ec2.instance
```

## Node Configuration

### Switch to Fuji Testnet

By default the Avalanche Node available through the AWS Marketplace syncs the Mainnet. If this is what you are looking for, you can skip this step.

For this tutorial you want to sync and validate the Fuji Testnet. Now that you're `ssh` ed into the EC2 instance you can make the required changes to sync Fuji instead of Mainnet.

First, confirm that the node is syncing the Mainnet by running the `info.getNetworkID` command.

#### `info.getNetworkID` Request

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNetworkID",
 "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

#### `info.getNetworkID` Response

The returned `networkID` will be 1 which is the network ID for Mainnet.

```
{
 "jsonrpc": "2.0",
 "result": {
 "networkID": "1"
 },
 "id": 1
}
```

Now you want to edit `/etc/avalanchego/conf.json` and change the `"network-id"` property from `"mainnet"` to `"fuji"`. To see the contents of `/etc/avalanchego/conf.json` you can `cat` the file.

```
cat /etc/avalanchego/conf.json
{
 "api-keystore-enabled": false,
 "http-host": "0.0.0.0",
 "log-dir": "/var/log/avalanchego",
 "db-dir": "/data/avalanchego",
 "api-admin-enabled": false,
 "public-ip-resolution-service": "opendns",
 "network-id": "mainnet"
}
```

Edit that `/etc/avalanchego/conf.json` with your favorite text editor and change the value of the `"network-id"` property from `"mainnet"` to `"fuji"`. Once that's complete, save the file and restart the Avalanche node via `sudo systemctl restart avalanchego`. You can then call the `info.getNetworkID` endpoint to confirm the change was successful.

#### `info.getNetworkID` Request

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNetworkID",
 "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

#### `info.getNetworkID` Response

The returned `networkID` will be 5 which is the network ID for Fuji.

```
{
 "jsonrpc": "2.0",
 "result": {
 "networkID": "5"
 },
 "id": 1
}
```

Next you run the `info.isBootstrapped` command to confirm if the Avalanche Validator node has finished bootstrapping.

#### `info.isBootstrapped` Request

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
```

```

"method": "info.isBootstrapped",
"params": {
 "chain": "P"
}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info

```

Once the node is finished bootstrapping, the response will be:

#### **info.isBootstrapped Response**

```
{
 "jsonrpc": "2.0",
 "result": {
 "isBootstrapped": true
 },
 "id": 1
}
```

Note that initially the response is `false` because the network is still syncing.

When you're adding your node as a Validator on the Avalanche Mainnet you'll want to wait for this response to return `true` so that you don't suffer from any downtime while validating. For this tutorial you're not going to wait for it to finish syncing as it's not strictly necessary.

#### **info.getNodeID Request**

Next, you want to get the NodeID which will be used to add the node as a Validator. To get the node's ID you call the `info.getNodeID` jsonrpc endpoint.

```

curl --location --request POST 'http://127.0.0.1:9650/ext/info' \
--header 'Content-Type: application/json' \
--data=raw '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID",
 "params": {}
}'

```

#### **info.getNodeID Response**

Take a note of the `nodeID` value which is returned as you'll need to use it in the next step when adding a validator via the Avalanche Web Wallet. In this case the `nodeID` is `NodeID-Q8Gfaaio9FAqCmZVExDq9bFvNPvDi7rt5`

```
{
 "jsonrpc": "2.0",
 "result": {
 "nodeID": "NodeID-Q8Gfaaio9FAqCmZVExDq9bFvNPvDi7rt5",
 "nodePOB": {
 "publicKey": "0x85675db18b326a9585bfd43892b25b71bf01b18587dc5fac136dc5343a9e8892cd6c49b0615ce928d53ff5dc7fd8945d",
 "proofOfPossession": "0x98a56f092830161243c1f1a613ad68a7f1fb25d2462ecf85065f22eaebb4e93a60e9e29649a32252392365d8f628b2571174f520331ee0063a94473f8db6888"
 }
 },
 "id": 1
}
```

## Add Node as Validator on Fuji via the Web Wallet

For adding the new node as a Validator on the Fuji testnet's Primary Network you can use the [Avalanche Web Wallet](#).

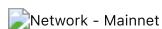


The Avalanche Web Wallet is a web-based application with no middleware or any kind of server communication. It can be either accessed online or compiled and run locally. The Avalanche Web Wallet is a multi-faceted jewel and offers validation/delegation, cross-chain transfers, reward estimation, asset/key management, and more.

#### **Switching the Connected Network**

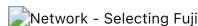
Check which network the wallet is connected to by looking at the top right of the screen. By default the Avalanche Web Wallet connects to Mainnet.

#### **Connected to Mainnet**



For the sake of this demo you want to connect the Wallet to the Fuji Testnet. At the top right of the wallet click "Mainnet" and from the nav menu select Fuji.

## Selecting Fuji



The wallet will display "Connecting..." while it is switching from Mainnet to Fuji.

## Connected to Fuji

Once the wallet has connected to Fuji a popup will display "Connected to Fuji"



## Connected to Fuji



You can follow the same steps for switching back to Mainnet from Fuji and for adding custom networks.

## The Earn Tab

To add a node as a Validator, first select the "Earn" tab in the left hand nav menu. Next click the "Add Validator" button.



### The Earn / Validate Form

Let's look at the input values for the `Earn / Validate` form.

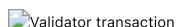
- Node ID: A unique ID derived from each individual node's staker certificate. Use the `NodeID` which was returned in the `info.getNodeID` response. In this example it's `NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5`
- Staking End Date: Your AVAX tokens will be locked until this date.
- Stake Amount: The amount of AVAX to lock for staking. On Mainnet the minimum required amount is 2,000 AVAX. On Testnet the minimum required amount is 1 AVAAX.
- Delegation Fee: You will claim this % of the rewards from the delegators on your node.
- Reward Address: A reward address is the destination address of the accumulated staking rewards.

Fill the fields and confirm! Carefully check the details, and click "Submit"!



### The AddValidatorTx Transaction

Once the transaction is successfully issued to the Avalanche Network the list of transactions in the right column will update with the new `AddValidatorTx` pushed to the top of the list. Click the magnifying glass icon and a new browser tab will open with the details of the `AddValidatorTx`. It will show details such as the total value of AVAX transferred, any AVAX which were burned, the blockchainID, the blockID, the NodeID of the validator, and the total time which has elapsed from the entire Validation period.



## Confirm That the Node is a Pending Validator on Fuji

As a last step you can call the `platform.getPendingValidators` endpoint to confirm that the Avalanche node which was recently spun up on AWS is no in the pending validators queue where it will stay for 5 minutes.

### platform.getPendingValidators Request

```
curl --location --request POST 'https://api.avax-test.network/ext/bc/P' \
--header 'Content-Type: application/json' \
--data-raw '{
 "jsonrpc": "2.0",
 "method": "platform.getPendingValidators",
 "params": {
 "subnetID": "11111111111111111111111111111111LpoYY",
 "nodeIDs": []
 },
 "id": 1
}'
```

### platform.getPendingValidators Response

```
{
 "jsonrpc": "2.0",
 "result": {
```

```

"validators": [
 {
 "txID": "4d7ZboCrND4FjnyNaF3qyosuGQsNeJ2R4KPJhHJ55VCU1Myjd",
 "startTime": "1673411918",
 "endTime": "1675313170",
 "stakeAmount": "1000000000",
 "nodeID": "NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5",
 "delegationFee": "2.0000",
 "connected": false,
 "delegators": null
 }
],
"delegators": []
},
"id": 1
}

```

You can also pass in the `NodeID` as a string to the `nodeIDs` array in the request body.

```

curl --location --request POST 'https://api.avax-test.network/ext/bc/P' \
--header 'Content-Type: application/json' \
--data-raw '{
 "jsonrpc": "2.0",
 "method": "platform.getPendingValidators",
 "params": {
 "subnetID": "11111111111111111111111111111111LpoYY",
 "nodeIDs": ["NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5"]
 },
 "id": 1
}'

```

This will filter the response by the `nodeIDs` array which will save you time by no longer requiring you to search through the entire response body for the `NodeIDs`.

```

{
 "jsonrpc": "2.0",
 "result": {
 "validators": [
 {
 "txID": "4d7ZboCrND4FjnyNaF3qyosuGQsNeJ2R4KPJhHJ55VCU1Myjd",
 "startTime": "1673411918",
 "endTime": "1675313170",
 "stakeAmount": "1000000000",
 "nodeID": "NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5",
 "delegationFee": "2.0000",
 "connected": false,
 "delegators": null
 }
],
 "delegators": []
 },
 "id": 1
}

```

After 5 minutes the node will officially start validating the Avalanche Fuji testnet and you will no longer see it in the response body for the `platform.getPendingValidators` endpoint. Now you will access it via the `platform.getCurrentValidators` endpoint.

#### **platform.getCurrentValidators Request**

```

curl --location --request POST 'https://api.avax-test.network/ext/bc/P' \
--header 'Content-Type: application/json' \
--data-raw '{
 "jsonrpc": "2.0",
 "method": "platform.getCurrentValidators",
 "params": {
 "subnetID": "11111111111111111111111111111111LpoYY",
 "nodeIDs": ["NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5"]
 },
 "id": 1
}'

```

#### **platform.getCurrentValidators Response**

```
{
 "jsonrpc": "2.0",
 "result": {
 "validators": [
 {
 "txID": "2hy57Z7Kiz8L3w2KonJJE1fs5j4JDzVHLjEALAHaXPr6VMMeDhk",
 "startTime": "1673411918",
 "endTime": "1675313170",
 "stakeAmount": "1000000000",
 "nodeID": "NodeID-Q8Gfaaio9FAqCmZVEXDq9bFvNPvDi7rt5",
 "rewardOwner": {
 "locktime": "0",
 "threshold": "1",
 "addresses": [
 "P-fujiltgj2c3k56enytw5d78rt0tsq31zg8wnftffwk7"
]
 },
 "validationRewardOwner": {
 "locktime": "0",
 "threshold": "1",
 "addresses": [
 "P-fujiltgj2c3k56enytw5d78rt0tsq31zg8wnftffwk7"
]
 },
 "delegationRewardOwner": {
 "locktime": "0",
 "threshold": "1",
 "addresses": [
 "P-fujiltgj2c3k56enytw5d78rt0tsq31zg8wnftffwk7"
]
 },
 "potentialReward": "5400963",
 "delegationFee": "2.0000",
 "uptime": "0.0000",
 "connected": false,
 "delegators": null
 }
],
 "id": 1
 }
}
```

## Mainnet

All of these steps can be applied to Mainnet. However, the minimum required Avax token amounts to become a validator is 2000 on the Mainnet. For more information, please read [this doc](#).

## Maintenance

AWS one click is meant to be used in automated environments, not as an end-user solution. You can still manage it manually, but it is not as easy as an Ubuntu instance or using the script:

- AvalancheGo binary is at `/usr/local/bin/avalanchego`
- Main node config is at `/etc/avalanchego/conf.json`
- Working directory is at `/home/avalanche/.avalanchego/` (and belongs to `avalanchego` user)
- Database is at `/data/avalanchego`
- Logs are at `/var/log/avalanchego`

For a simple upgrade you would need to place the new binary at `/usr/local/bin/`. If you run a Subnet, you would also need to place the VM binary into `/home/avalanche/.avalanchego/plugins`.

You can also look at using [this guide](#), but that won't address updating the Subnet, if you have one.

## Summary

**Avalanche is the first decentralized smart contracts platform built for the scale of global finance, with near-instant transaction finality. Now with an Avalanche Validator node available as a one-click install from the AWS Marketplace developers and entrepreneurs can on-ramp into the Avalanche ecosystem in a matter of minutes. If you have any questions or want to follow up in any way please join our Discord server at <https://chat.avax.network>. For more developer resources please check out our [Developer Documentation](#).**

**sidebar\_position: 2 description: The quickest way to learn about Avalanche is to run a node and interact with the network and geared toward people interested in how the Avalanche Platform works.**

# Run an Avalanche Node Manually

The quickest way to learn about Avalanche is to run a node and interact with the network.

In this tutorial, we will:

- Install and run an Avalanche node
- Connect to Avalanche

:::info

If you're interested in using a third-party service to host your node or run a validator, [check out the options](#).

:::

This tutorial is primarily geared toward developers and people interested in how the Avalanche Platform works. If you're just interested in setting up a node for staking, you may want to follow the [Set Up Avalanche Node With Installer](#) tutorial instead. Installer automates the installation process and sets it up as a system service, which is recommended for unattended operation. You may also try things out by following this tutorial first, and then later set up the node using the installer as a permanent solution.

## Requirements

### Computer Hardware and OS

Avalanche is an incredibly lightweight protocol, so nodes can run on commodity hardware. Note that as network usage increases, hardware requirements may change.

- CPU: Equivalent of 8 AWS vCPU
- RAM: 16 GiB
- Storage: 1 TiB SSD
- OS: Ubuntu 20.04 or MacOS >= 12

:::caution

Please do not try running a node on an HDD, as you may get poor and random read/write latencies, therefore reducing performance and reliability.

:::

### Networking

To run successfully, AvalancheGo needs to accept connections from the Internet on the network port `9651`. Before you proceed with the installation, you need to determine the networking environment your node will run in.

#### Running on a Cloud Provider

If your node is running on a cloud provider computer instance, it will have a static IP. Find out what that static IP is, or set it up if you didn't already.

#### Running on a Home Connection

If you're running a node on a computer that is on a residential internet connection, you have a dynamic IP; that is, your IP will change periodically. You will need to set up inbound port forwarding of port `9651` from the internet to the computer the node is installed on.

As there are too many models and router configurations, we cannot provide instructions on what exactly to do, but there are online guides to be found (like [this](#), or [this](#)), and your service provider support might help too.

:::warning

Please note that a fully connected Avalanche node maintains and communicates over a couple of thousand of live TCP connections. For some low-powered and older home routers that might be too much to handle. If that is the case you may experience lagging on other computers connected to the same router, node getting benched, failing to sync and similar issues.

:::

## Run an Avalanche Node

Let's install AvalancheGo, the Golang implementation of an Avalanche node, and connect to the Avalanche primary network.

### Download AvalancheGo

The node is a binary program. You can either download the source code and then build the binary program, or you can download the pre-built binary. You don't need to do both.

Downloading [pre-built binary](#) is easier and recommended if you're just looking to run your own node and stake on it.

Building the node from source is recommended if you're a developer looking to experiment and build on Avalanche.

#### Source Code

First install Go 1.19.6 or later. Follow the instructions [here](#). You can verify by running `go version`.

Set `$GOPATH` environment variable properly for Go to look for Go Workspaces. Please read [this](#) for details. You can verify by running `echo $GOPATH`.

Download the AvalancheGo repository into your `$GOPATH`:

```
cd $GOPATH
mkdir -p src/github.com/ava-labs
cd src/github.com/ava-labs
git clone https://github.com/ava-labs/avalanchego.git
cd avalanchego
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchego.git
```

You can find more about SSH and how to use it [here](#). :::

Note: This checkouts to the master branch. For the latest stable version, checkout the latest tag.

Build AvalancheGo:

```
./scripts/build.sh
```

The binary, named `avalanchego`, is in `avalanchego/build`. If you've followed the instructions so far, this will be within your `$GOPATH` at:  
`$GOPATH/src/github.com/ava-labs/avalanchego/build`.

To begin running AvalancheGo, run the following (hit Ctrl+C to stop your node):

```
./build/avalanchego
```

## Binary

If you want to download a pre-built binary instead of building it yourself, go to our [releases page](#), and select the release you want (probably the latest one.)

Under `Assets`, select the appropriate file.

For MacOS: Download: `avalanchego-macos-<VERSION>.zip` Unzip: `unzip avalanchego-macos-<VERSION>.zip` The resulting folder, `avalanchego-<VERSION>`, contains the binaries.

For Linux on PCs or cloud providers: Download: `avalanchego-linux-amd64-<VERSION>.tar.gz` Unzip: `tar -xvf avalanchego-linux-amd64-<VERSION>.tar.gz` The resulting folder, `avalanchego-<VERSION>-linux`, contains the binaries.

For Linux on Arm64-based computers: Download: `avalanchego-linux-arm64-<VERSION>.tar.gz` Unzip: `tar -xvf avalanchego-linux-arm64-<VERSION>.tar.gz` The resulting folder, `avalanchego-<VERSION>-linux`, contains the binaries.

## Start a Node, and Connect to Avalanche

If you built from source:

```
./build/avalanchego
```

If you are using the pre-built binaries on MacOS:

```
./avalanchego-<VERSION>/build/avalanchego
```

If you are using the pre-built binaries on Linux:

```
./avalanchego-<VERSION>-linux/avalanchego
```

By default (without specifying `--network-id`), this node will connect to the Mainnet which may take much longer time to finish bootstrapping. See [this](#) for connecting to Fuji Testnet.

When the node starts, it has to bootstrap (catch up with the rest of the network). You will see logs about bootstrapping. When a given chain is done bootstrapping, it prints logs like this:

```
[09-09|17:01:45.295] INFO <C Chain> snowman/transitive.go:392 consensus starting {"lastAcceptedBlock": "2qaFwDJtmCCbMKP4jRpJwH8EFws82Q2yClHhWgAiy3tGrpGFeb"}
[09-09|17:01:46.199] INFO <P Chain> snowman/transitive.go:392 consensus starting {"lastAcceptedBlock": "2ofmPJUWZbdroCP EMv6aHGvZ45oa8SBp2reEm9gN xvFjnfSGFP"}
[09-09|17:01:51.628] INFO <X Chain> snowman/transitive.go:334 consensus starting {"lenFrontier": 1}
```

To check if a given chain is done bootstrapping, in another terminal window call `info.isBootstrapped` by copying and pasting the following command:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.isBootstrapped",
 "params": {
 "chain": "X"
 }
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

If this returns `true`, the chain is bootstrapped; otherwise, it returns `false`. If you make other API calls to a chain that is not done bootstrapping, it will return `API call rejected because chain is not done bootstrapping`. If you are still experiencing issues please contact us on [Discord](#).

Your node is running and connected now. If you want to use your node as a validator on the main net, check out [this tutorial](#) to find out how to add your node as a validator using the web wallet.

You can use `Ctrl + C` to kill the node.

To be able to make API calls to your node from other machines, when starting up the node include argument `--http-host=` (for example `./build/avalanchego --http-host=`)

#### Connect to Fuji Testnet

To connect to the Fuji Testnet instead of the Mainnet, use argument `--network-id=fuji`. You can get funds on the Testnet from the [faucet](#).

### What Next?

Now that you've launched your Avalanche node, what should you do next?

Your Avalanche node will perform consensus on its own, but it is not yet a validator on the network. This means that the rest of the network will not query your node when sampling the network during consensus. If you want to add your node as a validator, check out [Add a Validator](#) to take it a step further.

Also check out the [Maintain](#) section to learn about how to maintain and customize your node to fit your needs.

#### sidebar\_position: 6 description: GCP Validator Provisioning

## Run an Avalanche Node with Google Cloud Platform

:::caution This document was written by a community member, some information may be out of dated. :::

### Introduction

Google's Cloud Platform (GCP) is a scalable, trusted and reliable hosting platform. Google operates a significant amount of its own global networking infrastructure. It's [fiber network](#) is **huge** and it can provide highly stable and consistent global connectivity. In this article we will leverage GCP to deploy a node on which Avalanche can be installed via [terraform](#). Leveraging `terraform` may seem like overkill, but speaking as someone who has managed extremely large compute estates for over a decade I believe it will set you apart as an operator and administrator as it will enable you greater flexibility and provide the basis on which you can easily build further automation.

### Conventions

- Items highlighted in this manner are GCP parlance and can be searched for further reference in the Google documentation for their cloud products.

### Important Notes

- The machine type used in this documentation is for reference only and the actual sizing you use will depend entirely upon the amount that is staked and delegated to the node.

### Architectural Description

This section aims to describe the architecture of the system that the steps in the [Setup Instructions](#) section deploy when enacted. This is done so that the executor can not only deploy the reference architecture, but also understand and potentially optimize it for their needs.

### Project

We will create and utilize a single GCP `Project` for deployment of all resources.

### Service Enablement

Within our GCP project we will need to enable the following Cloud Services:

- `Compute Engine`
- `IAP`

### Networking

#### Compute Network

We will deploy a single `Compute Network` object. This unit is where we will deploy all subsequent networking objects. It provides a logical boundary and securitization context should you wish to deploy other chain stacks or other infrastructure in GCP.

#### Public IP

Avalanche requires that a validator communicate outbound on the same public IP address that it advertises for other peers to connect to it on. Within GCP this precludes the possibility of us using a Cloud NAT Router for the outbound communications and requires us to bind the public IP that we provision to the interface of the machine. We will provision a single `EXTERNAL static IPv4 Compute Address`.

#### Subnets

For the purposes of this documentation we will deploy a single `Compute Subnetwork` in the US-EAST1 Region with a /24 address range giving us 254 IP addresses (not all usable but for the sake of generalized documentation).

## Compute

### Disk

We will provision a single 400GB `PD-SSD` disk that will be attached to our VM.

### Instance

We will deploy a single `Compute Instance` of size `e2-standard-8`. Observations of operations using this machine specification suggest it is memory over provisioned and could be brought down to 16GB using custom machine specification; but please review and adjust as needed (the beauty of compute virtualization!!).

### Zone

We will deploy our instance into the `US-EAST1-B` Zone

### Firewall

We will provision the following `Compute Firewall` rules:

- IAP INGRESS for SSH (TCP 22) - this only allows GCP IAP sources inbound on SSH.
- P2P INGRESS for AVAX Peers (TCP 9651)

These are obviously just default ports and can be tailored to your needs as you desire.

## Setup Instructions

### GCP Account

1. If you don't already have a GCP account go create one [here](#)

You will get some free bucks to run a trial, the trial is feature complete but your usage will start to deplete your free bucks so turn off anything you don't need and/or add a credit card to your account if you intend to run things long term to avoid service shutdowns.

### Project

Login to the GCP `Cloud Console` and create a new `Project` in your organization. Let's use the name `my-avax-nodes` for the sake of this setup.

1. Select Project Dropdown
2. Click New Project Button
3. Create New Project

### Terraform State

Terraform uses a state files to compose a differential between current infrastructure configuration and the proposed plan. You can store this state in a variety of different places, but using GCP storage is a reasonable approach given where we are deploying so we will stick with that.

1. Select Cloud Storage Browser
2. Create New Bucket

Authentication to GCP from terraform has a few different options which are laid out [here](#). Please chose the option that aligns with your context and ensure those steps are completed before continuing.

:::note

Depending upon how you intend to execute your terraform operations you may or may not need to enable public access to the bucket. Obviously, not exposing the bucket for `public` access (even if authenticated) is preferable. If you intend to simply run terraform commands from your local machine then you will need to open the access up. I recommend to employ a full CI/CD pipeline using GCP Cloud Build which if utilized will mean the bucket can be marked as `private`. A full walk through of Cloud Build setup in this context can be found [here](#)

:::

### Clone GitHub Repository

I have provided a rudimentary terraform construct to provision a node on which to run Avalanche which can be found [here](#). Documentation below assumes you are using this repository but if you have another terraform skeleton similar steps will apply.

### Terraform Configuration

1. If running terraform locally, please [install](#) it.
2. In this repository, navigate to the `terraform` directory.
3. Under the `projects` directory, rename the `my-avax-project` directory to match your GCP project name that you created (not required, but nice to be consistent)
4. Under the folder you just renamed locate the `terraform.tfvars` file.
5. Edit this file and populate it with the values which make sense for your context and save it.
6. Locate the `backend.tf` file in the same directory.
7. Edit this file ensuring to replace the `bucket` property with the GCS bucket name that you created earlier.

If you do not wish to use cloud storage to persist terraform state then simply switch the `backend` to some other desirable provider.

## Terraform Execution

Terraform enables us to see what it would do if we were to run it without actually applying any changes... this is called a `plan` operation. This plan is then enacted (optionally) by an `apply`.

### Plan

1. In a terminal which is able to execute the `tf` binary, `cd` to the `~/.my-avax-project` directory that you renamed in step 3 of Terraform Configuration`.
2. Execute the command `tf plan`
3. You should see a JSON output to the stdout of the terminal which lays out the operations that terraform will execute to apply the intended state.

### Apply

1. In a terminal which is able to execute the `tf` binary, `cd` to the `~/.my-avax-project` directory that you renamed in step 3 of Terraform Configuration`.
2. Execute the command `tf apply`

If you want to ensure that terraform does **exactly** what you saw in the `apply` output, you can optionally request for the `plan` output to be saved to a file to feed to `apply`. This is generally considered best practice in highly fluid environments where rapid change is occurring from multiple sources.

## Conclusion

**Establishing CI/CD practices using tools such as GitHub and Terraform to manage your infrastructure assets is a great way to ensure base disaster recovery capabilities and to ensure you have a place to embed any ~tweaks you have to make operationally removing the potential to miss them when you have to scale from 1 node to 10. Having an automated pipeline also gives you a place to build a bigger house... what starts as your interest in building and managing a single AVAX node today can quickly change into you building an infrastructure operation for many different chains working with multiple different team members. I hope this may have inspired you to take a leap into automation in this context!**

**sidebar\_position: 9 description: Detailed instructions for running an Avalanche node with Latitude.sh**

## Run an Avalanche Node with Latitude.sh

### Introduction

This tutorial will guide you through setting up an Avalanche node on [Latitude.sh](#). Latitude.sh provides high-performance lightning-fast bare metal servers to ensure that your node is highly secure, available, and accessible.

To get started, you'll need:

- A Latitude.sh account
- A terminal with which to SSH into your Latitude.sh machine

For the instructions on creating an account and server with Latitude.sh, please reference their [GitHub tutorial](#), or visit [this page](#) to sign up and create your first project.

This tutorial assumes your local machine has a Unix-style terminal. If you're on Windows, you'll have to adapt some of the commands used here.

### Configuring Your Server

#### Create a Latitude.sh Account

At this point your account has been verified, and you have created a new project and deployed the server according to the instructions linked above.

#### Access Your Server & Further Steps

All your Latitude.sh credentials are available by clicking the `server` under your project, and can be used to access your Latitude.sh machine from your local machine using a terminal.

:::note You will need to install the `avalanche node` installer script directly in the server's terminal.:::

After gaining access, we'll need to set up our Avalanche node. To do this, follow the instructions here to install and run your node [Set Up Avalanche Node With Installer](#).

Your AvalancheGo node should now be running and in the process of bootstrapping, which can take a few hours. To check if it's done, you can issue an API call using `curl`. The request is:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.isBootstrapped",
 "params": {
 "chain": "X"
}' http://127.0.0.1:4001
```

```
}
```

```
} -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Once the node is finished bootstrapping, the response will be:

```
{
 "jsonrpc": "2.0",
 "result": {
 "isBootstrapped": true
 },
 "id": 1
}
```

You can continue on, even if AvalancheGo isn't done bootstrapping. In order to make your node a validator, you'll need its node ID. To get it, run:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

The response contains the node ID.

```
{
 "jsonrpc": "2.0",
 "result": { "nodeID": "KhDnAoZDW8iRJ3F26iQgK5xXVFMPcaYeu" },
 "id": 1
}
```

In the above example the node ID is `NodeID-KhDnAoZDW8iRJ3F26iQgK5xXVFMPcaYeu`.

AvalancheGo has other APIs, such as the [Health API](#), that may be used to interact with the node. Some APIs are disabled by default. To enable such APIs, modify the ExecStart section of `/etc/systemd/system/avalanchego.service` (created during the installation process) to include flags that enable these endpoints. Don't manually enable any APIs unless you have a reason to.

Exit out of the SSH server by running:

```
exit
```

## Upgrading Your Node

AvalancheGo is an ongoing project and there are regular version upgrades. Most upgrades are recommended but not required. Advance notice will be given for upgrades that are not backwards compatible. To update your node to the latest version, SSH into your server using a terminal and run the installer script again.

```
./avalanchego-installer.sh
```

Your machine is now running the newest AvalancheGo version. To see the status of the AvalancheGo service, run `sudo systemctl status avalanchego`.

## Wrap Up

**That's it! You now have an AvalancheGo node running on a Latitude.sh machine. We recommend setting up [node monitoring](#) for your AvalancheGo node.**

**sidebar\_position: 5 description: Running a validator and staking with Avalanche provides extremely competitive rewards of between 9% and 11% depending on the length you stake for. Find out more info here.**

## Run an Avalanche Node with Microsoft Azure

:::caution This document was written by a community member, some information may be out of date. :::

Running a validator and staking with Avalanche provides extremely competitive rewards of between 9.69% and 11.54% depending on the length you stake for. The maximum rate is earned by staking for a year, whilst the lowest rate for 14 days. There is also no slashing, so you don't need to worry about a hardware failure or bug in the client which causes you to lose part or all of your stake. Instead with Avalanche you only need to currently maintain at least 80% uptime to receive rewards. If you fail to meet this requirement you don't get slashed, but you don't receive the rewards. **You also do not need to put your private keys onto a node to begin validating on that node.** Even if someone breaks into your cloud environment and gains access to the node, the worst they can do is turn off the node.

Not only does running a validator node enable you to receive rewards in AVAX, but later you will also be able to validate other Subnets in the ecosystem as well and receive rewards in the token native to their Subnets.

Hardware requirements to run a validator are relatively modest: 8 CPU cores, 16 GB of RAM and 1 TB SSD. It also doesn't use enormous amounts of energy. Avalanche's [revolutionary consensus mechanism](#) is able to scale to millions of validators participating in consensus at once, offering unparalleled decentralisation.

Currently the minimum amount required to stake to become a validator is 2,000 AVAX (which can be reduced over time as price increases). Alternatively, validators can also charge a small fee to enable users to delegate their stake with them to help towards running costs. You can use a calculator [here](#) to see how much rewards you would earn when running a node, compared to delegating.

In this article we will step through the process of configuring a node on Microsoft Azure. This tutorial assumes no prior experience with Microsoft Azure and will go through each step with as few assumptions possible.

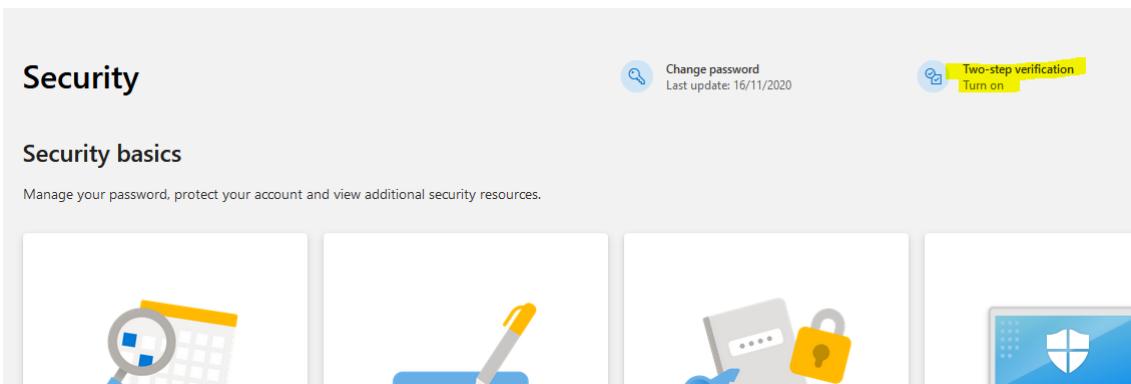
At the time of this article, spot pricing for a virtual machine with 2 Cores and 8 GB memory costs as little as \$0.01060 per hour which works out at about \$113.44 a year, **a saving of 83.76% compared to normal pay as you go prices**. In comparison a virtual machine in AWS with 2 Cores and 4 GB Memory with spot pricing is around \$462 a year.

## Initial Subscription Configuration

### Set up 2 Factor

First you will need a Microsoft Account, if you don't have one already you will see an option to create one at the following link. If you already have one, make sure to set up 2 Factor authentication to secure your node by going to the following link and then selecting "Two-step verification" and following the steps provided.

<https://account.microsoft.com/security>



Once two factor has been configured log into the Azure portal by going to <https://portal.azure.com> and signing in with your Microsoft account. When you login you won't have a subscription, so we need to create one first. Select "Subscriptions" as highlighted below:

 Microsoft Azure  Search resources, services, and

## Welcome to Azure!

Don't have a subscription? Check out the following options.



### Start with an Azure free trial

Get \$200 free credit toward Azure products and services, plus 12 months of popular [free services](#).

[Start](#) [Learn more](#)

### Azure services



Then select "+ Add" to add a new subscription

 Microsoft Azure

Home >

## Subscriptions

 [Add](#)

View list of subscriptions for which you have role-based access  
Showing subscriptions in directory. Don't see a subscription?

My role 

8 selected

[Apply](#)

Showing 0 of 0 subscriptions  Show only subscriptions selected

 [Search](#)

Subscription name ↑↓

You don't have any subscriptions

If you want to use Spot Instance VM Pricing (which will be considerably cheaper) you can't use a Free Trial account (and you will receive an error upon validation), so **make sure to select Pay-As-You-Go**.

## Select an offer for your subscription

PREVIEW

^ Most Popular Offers

### Free Trial

Full access to all services. Explore any service that you want. [Learn more](#)

Select offer

### Pay-As-You-Go

This flexible pay-as-you-go plan involves no up-front costs, and no long term commitment. You pay only for the resources that you use. [Learn more](#)

Select offer

Enter your billing details and confirm identity as part of the sign-up process, when you get to Add technical support select the without support option (unless you want to pay extra for support) and press Next.

#### 1 Your profile

#### 2 Identity verification by phone

#### 3 Payment Information

#### 4 Add technical support

For one-on-one technical support, you'll need a [support plan](#). Whether you're new to Azure or already building business-critical applications, our support plans provide expert guidance from Azure engineers who will work with you to prevent and solve problems. By upgrading to a plan you agree to the [offer details](#).

**Getting started on Azure - Developer plan £21.61/month**

For developers or teams looking to quickly and effectively get started on Azure, technical support is available weekdays from 9:00 AM to 5:00 PM with initial response times under 8 business hours.

**Production use of Azure - Standard plan £74.53/month**

For teams running production applications, get 24x7 technical support and fast initial response times under 2 hours.

**Business-critical use of Azure - Professional Direct plan £745.31/month**

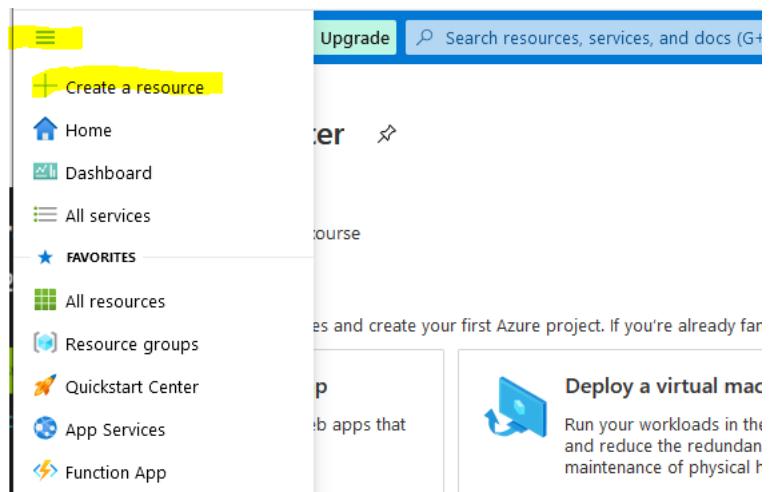
For business-critical applications, a cloud advisor provides guidance and advocacy to help improve reliability and optimize costs. You also get 24x7 technical support with the fastest initial response times under 1 hour.

No technical support, or I am already covered through Microsoft Premier support.

Next

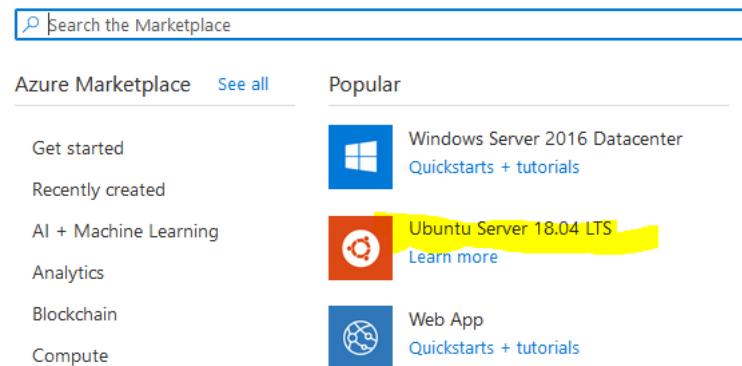
## Create a Virtual Machine

Now that we have a subscription, we can create the Ubuntu Virtual Machine for our Avalanche Node. Select the icon in the top left for the Menu and choose "+ Create a resource"



The screenshot shows the Azure portal interface. On the left, there's a sidebar with various navigation options like Home, Dashboard, All services, Favorites, All resources, Resource groups, Quickstart Center, App Services, and Function App. A yellow box highlights the "Create a resource" button at the top of this sidebar. At the top right, there's an "Upgrade" button and a search bar with the placeholder "Search resources, services, and docs (G+)". Below the search bar, there's a section titled "Deploy a virtual machine" with a sub-section "Ubuntu Server 18.04 LTS".

Select Ubuntu Server 18.04 LTS (this will normally be under the popular section or alternatively search for it in the marketplace)



The screenshot shows the Azure Marketplace page. On the left, there's a sidebar with "Azure Marketplace" and a "See all" link. Below it are categories: Get started, Recently created, AI + Machine Learning, Analytics, Blockchain, and Compute. On the right, there's a "Popular" section with several items: Windows Server 2016 Datacenter, Ubuntu Server 18.04 LTS (which has a yellow box around its name and a blue "Learn more" button), and Web App. Each item has a small icon and a "Quickstarts + tutorials" link.

This will take you to the Create a virtual machine page as shown below:

## Create a virtual machine

Basics Disks Networking Management Advanced Tags Review + create

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. [Learn more](#)

### Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

|                  |                                                                              |
|------------------|------------------------------------------------------------------------------|
| Subscription *   | <input type="text" value="Pay-As-You-Go"/>                                   |
| Resource group * | <input type="text" value="(New) Resource group"/> <a href="#">Create new</a> |

### Instance details

|                        |                                                                                                                  |
|------------------------|------------------------------------------------------------------------------------------------------------------|
| Virtual machine name * | <input type="text"/>                                                                                             |
| Region *               | <input type="text" value="(US) East US"/>                                                                        |
| Availability options   | <input type="text" value="No infrastructure redundancy required"/>                                               |
| Image *                | <input type="text" value="Ubuntu Server 18.04 LTS - Gen1"/> <a href="#">Browse all public and private images</a> |
| Azure Spot instance    | <input type="radio"/> Yes <input checked="" type="radio"/> No                                                    |
| Size *                 | <input type="text" value="Standard_D2s_v3 - 2 vcpus, 8 GiB memory (£52.23/month)"/> <a href="#">Select size</a>  |

### Administrator account

|                                                                                                                                                                                                                             |                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Authentication type                                                                                                                                                                                                         | <input checked="" type="radio"/> SSH public key <input type="radio"/> Password |
| <p><span style="color: #0078d4;">i</span> Azure now automatically generates an SSH key pair for you and allows you to store it for future use. It is a fast, simple, and secure way to connect to your virtual machine.</p> |                                                                                |
| Username *                                                                                                                                                                                                                  | <input type="text" value="azureuser"/>                                         |
| SSH public key source                                                                                                                                                                                                       | <input type="text" value="Generate new key pair"/>                             |
| Key pair name *                                                                                                                                                                                                             | <input type="text" value="Name the SSH public key"/>                           |

### Inbound port rules

Select which virtual machine network ports are accessible from the public internet. You can specify more limited or granular network access on the Networking tab.

|                        |                                                                                  |
|------------------------|----------------------------------------------------------------------------------|
| Public inbound ports * | <input type="radio"/> None <input checked="" type="radio"/> Allow selected ports |
| Select inbound ports * | <input type="text" value="SSH (22)"/>                                            |

[Review + create](#)

[< Previous](#)

[Next : Disks >](#)

First, enter a virtual machine a name, this can be anything but in my example, I have called it Avalanche (This will also automatically change the resource group name to match)

Then select a region from the drop-down list. Select one of the recommended ones in a region that you prefer as these tend to be the larger ones with most features enabled and cheaper prices. In this example I have selected North Europe.

## Instance details

|                        |                                                                                          |
|------------------------|------------------------------------------------------------------------------------------|
| Virtual machine name * | <input type="text" value="Avalanche"/> <span style="color: green;">✓</span>              |
| Region *               | <input type="text" value="(Europe) North Europe"/> <span style="color: yellow;">^</span> |
| Availability options   | <input type="text"/>                                                                     |
| Image *                | <input type="text"/> Recommended <span style="color: blue;">?</span>                     |
| Azure Spot instance    | <input type="checkbox"/> (US) East US                                                    |
| Eviction type          | <input type="checkbox"/> (US) East US 2                                                  |
|                        | <input type="checkbox"/> (US) South Central US                                           |
|                        | <input type="checkbox"/> (US) West US 2                                                  |
|                        | <input type="checkbox"/> (Asia Pacific) Australia East                                   |
|                        | <input type="checkbox"/> (Asia Pacific) Southeast Asia                                   |
| Eviction policy        | <input type="checkbox"/> (Europe) North Europe                                           |
|                        | <input type="checkbox"/> (Europe) UK South                                               |

You have the option of using spot pricing to save significant amounts on running costs. Spot instances use a supply and demand market price structure. As demand for instances goes up, the price for the spot instance goes up. If there is insufficient capacity, then your VM will be turned off. The chances of this happening are incredibly low though, especially if you select the Capacity only option. Even in the unlikely event it does get turned off temporarily you only need to maintain at least 80% up time to receive the staking rewards and there is no slashing implemented in Avalanche.

Select Yes for Azure Spot instance, select Eviction type to Capacity Only and **make sure to set the eviction policy to Stop / Deallocate. This is very important otherwise the VM will be deleted**

Basics Disks Networking Management Advanced Tags Review + create

Create a virtual machine that runs Linux or Windows. Select an image from Azure marketplace or use your own customized image. Complete the Basics tab then Review + create to provision a virtual machine with default parameters or review each tab for full customization. [Learn more](#)

#### Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ  ▼

Resource group \* ⓘ  ▼  
[Create new](#)

#### Instance details

Virtual machine name \* ⓘ  ✓

Region \* ⓘ  ▼

Availability options ⓘ  ▼

Image \* ⓘ  ▼  
[Browse all public and private images](#)

Azure Spot instance ⓘ  Yes  No

Eviction type ⓘ  Capacity only: evict virtual machine when Azure needs the capacity for pay as you go workloads. Your max price is set to the pay as you go rate.  
 Price or capacity: choose a max price and Azure will evict your virtual machine when the cost of the instance is greater than your max price or when Azure needs the capacity for pay as you go workloads.

Eviction policy ⓘ  Stop / Deallocate  Delete

Size \* ⓘ  ▼  
[Select size](#)  
[View pricing history and compare prices in nearby regions](#)

Choose "Select size" to change the Virtual Machine size, and from the menu select D2s\_v4 under the D-Series v4 selection (This size has 2 Cores, 8 GB Memory and enables Premium SSDs). You can use F2s\_v2 instances instead, with are 2 Cores, 4 GB Memory and enables Premium SSDs) but the spot price actually works out cheaper for the larger VM currently with spot instance prices. You can use [this link](#) to view the prices across the different regions.

| The latest generation D family sizes recommended for your general purpose needs |                 |   |    |    |       |     |
|---------------------------------------------------------------------------------|-----------------|---|----|----|-------|-----|
| D-Series v4                                                                     |                 | 2 | 8  | 4  | 3200  | 16  |
| D2as_v4                                                                         | General purpose | 2 | 8  | 4  | 3200  | 16  |
| D2ds_v4                                                                         | General purpose | 2 | 8  | 4  | 3200  | 75  |
| D2s_v4                                                                          | General purpose | 2 | 8  | 4  | 3200  | 0   |
| D4as_v4                                                                         | General purpose | 4 | 16 | 8  | 6400  | 32  |
| D4ds_v4                                                                         | General purpose | 4 | 16 | 8  | 6400  | 150 |
| D4s_v4                                                                          | General purpose | 4 | 16 | 8  | 6400  | 0   |
| D8as_v4                                                                         | General purpose | 8 | 32 | 16 | 12800 | 64  |
| D8ds_v4                                                                         | General purpose | 8 | 32 | 16 | 12800 | 300 |

[Select](#)

Once you have selected the size of the Virtual Machine, select "View pricing history and compare prices in nearby regions" to see how the spot price has changed over the last 3 months, and whether it's cheaper to use a nearby region which may have more spare capacity.

Eviction policy [\(i\)](#)

Stop / Deallocate  Delete

Size \* [\(i\)](#)

Standard\_D2s\_v4 - (\$0.01295/hour) [\(v\)](#)

Select size

[View pricing history and compare prices in nearby regions](#)

At the time of this article, spot pricing for D2s\_v4 in North Europe costs \$0.07975 per hour, or around \$698.61 a year. With spot pricing, the price falls to \$0.01295 per hour, which works out at about \$113.44 a year, a **saving of 83.76%**!

There are some regions which are even cheaper, East US for example is \$0.01060 per hour or around \$92.86 a year!



Maximum price you want to pay per hour (USD) [\(i\)](#)



Below you can see the price history of the VM over the last 3 months for North Europe and regions nearby.



Maximum price you want to pay per hour (USD) [\(i\)](#)

**0.07975**

#### Cheaper Than Amazon AWS

As a comparison a c5.large instance costs \$0.085 USD per hour on AWS. This totals ~\$745 USD per year. Spot instances can save 62%, bringing that total down to \$462.

The next step is to change the username for the VM, to align with other Avalanche tutorials change the username to Ubuntu. Otherwise you will need to change several commands later in this article and swap out Ubuntu with your new username.

## Create a virtual machine

Region \* ⓘ (Europe) North Europe

Availability options ⓘ No infrastructure redundancy required

Image \* ⓘ Ubuntu Server 18.04 LTS - Gen1  
Browse all public and private images

Azure Spot instance ⓘ Yes  No

Eviction type ⓘ Capacity only: evict virtual machine when Azure needs the capacity for pay as you go workloads. Your max price is set to the pay as you go rate.  
Price or capacity: choose a max price and Azure will evict your virtual machine when the cost of the instance is greater than your max price or when Azure needs the capacity for pay as you go workloads.

Eviction policy ⓘ Stop / Deallocate  Delete

Size \* ⓘ Standard\_D2s\_v4 - (\$0.01295/hour)  
Select size  
View pricing history and compare prices in nearby regions

Maximum price you want to pay per hour (USD) ⓘ 0.07975  
Enter a price greater than or equal to the hardware costs (\$0.01295)

Administrator account

Authentication type ⓘ SSH public key  Password

**Info** Azure now automatically generates an SSH key pair for you and allows you to store it for future use. It is a fast, simple, and secure way to connect to your virtual machine.

Username \* ⓘ ubuntu

SSH public key source ⓘ Generate new key pair

Key pair name \* ⓘ Avalanche\_key

Inbound port rules

Select which virtual machine network ports are accessible from the public internet. You can specify more limited or granular network access on the Networking tab.

Public inbound ports \* ⓘ None  Allow selected ports

Select inbound ports \* ⓘ SSH (22)

**Warning** This will allow all IP addresses to access your virtual machine. This is only recommended for testing. Use the Advanced controls in the Networking tab to create rules to limit inbound traffic to known IP addresses.

**Review + create** < Previous **Next : Disks >**

### Disks

Select Next: Disks to then configure the disks for the instance. There are 2 choices for disks, either Premium SSD which offer greater performance with a 64 GB disk costs around \$10 a month, or there is the standard SSD which offers lower performance and is around \$5 a month. You also have to pay \$0.002 per 10,000

transaction units (reads / writes and deletes) with the Standard SSD, whereas with Premium SSDs everything is included. Personally, I chose the Premium SSD for greater performance, but also because the disks are likely to be heavily used and so may even work out cheaper in the long run.

Select Next: Networking to move onto the network configuration

## Create a virtual machine

Basics   Disks   Networking   Management   Advanced   Tags   Review + create

Azure VMs have one operating system disk and a temporary disk for short-term storage. You can attach additional data disks. The size of the VM determines the type of storage you can use and the number of data disks allowed. [Learn more](#)

**Disk options**

OS disk type \* ⓘ

Encryption type \*

Enable Ultra Disk compatibility ⓘ  Yes  No  
Ultra disk is available only for Availability Zones in northeurope.

**Data disks**

You can add and configure additional data disks for your virtual machine or attach existing disks. This VM also comes with a temporary disk.

| LUN | Name | Size (GiB) | Disk type | Host caching |
|-----|------|------------|-----------|--------------|
|     |      |            |           |              |

[Create and attach a new disk](#)   [Attach an existing disk](#)

[Review + create](#)   [< Previous](#)   [Next : Networking >](#)

## Network Config

You want to use a Static IP so that the public IP assigned to the node doesn't change in the event it stops. Under Public IP select "Create new"

Basics   Disks   **Networking**   Management   Advanced   Tags   Review + create

Define network connectivity for your virtual machine by configuring network interface card (NIC) settings. You can control ports, inbound and outbound connectivity with security group rules, or place behind an existing load balancing solution.

[Learn more](#)

### Network interface

When creating a virtual machine, a network interface will be created for you.

Virtual network \* ⓘ    
[Create new](#)

Subnet \* ⓘ

Public IP ⓘ    
[Create new](#)

Then select "Static" as the Assignment type

## Create public IP address X

Name \*

SKU (i)  
 Basic  Standard

Assignment  
 Dynamic  Static

Then we need to configure the network security group to control access inbound to the Avalanche node. Select "Advanced" as the NIC network security group type and select "Create new"

|                                                                                 |                                                                                                  |
|---------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Subnet <span style="color: red;">*</span> <span style="color: blue;">(i)</span> | <input type="text" value="(new) default (10.0.0.0/24)"/>                                         |
| Public IP <span style="color: blue;">(i)</span>                                 | <input type="text" value="(new) Avalanche-ip"/><br><a href="#">Create new</a>                    |
| NIC network security group <span style="color: blue;">(i)</span>                | <input type="radio"/> None <input type="radio"/> Basic <input checked="" type="radio"/> Advanced |
| Configure network security group <span style="color: red;">*</span>             | <input type="text" value="(new) Avalanche-nsg"/><br><a href="#">Create new</a>                   |

For security purposes you want to restrict who is able to remotely connect to your node. To do this you will first want to find out what your existing public IP is. This can be done by going to google and searching for "what's my IP"

Google search results for "what's my ip":  
what's my ip  
All News Books Shopping  
About 1,850,000,000 results (0.47 seconds)  
What's my IP  
143.24. [REDACTED]  
Your public IP address

It's likely that you have been assigned a dynamic public IP for your home, unless you have specifically requested it, and so your assigned public IP may change in the future. It's still recommended to restrict access to your current IP though, and then in the event your home IP changes and you are no longer able to remotely connect to the VM, you can just update the network security rules with your new public IP so you are able to connect again.

NOTE: If you need to change the network security group rules after deployment if your home IP has changed, search for "avalanche-nsg" and you can modify the rule for SSH and Port 9650 with the new IP. **Port 9651 needs to remain open to everyone** though as that's how it communicates with other Avalanche nodes.

CloudFormation search results for "avalanche-nsg":  
Services: No results were found.  
Resources: Avalanche-nsg (Network security group)

Now that you have your public IP select the default allow ssh rule on the left under inbound rules to modify it. Change Source from "Any" to "IP Addresses" and then enter in your Public IP address that you found from google in the Source IP address field. Change the Priority towards the bottom to 100 and then press Save.

The screenshot shows the Azure portal interface for creating a network security group. On the left, there's a sidebar with navigation links: Home > New > Ubuntu Server 18.04 LTS > Create a virtual machine. Below this, it says 'Create network security group'. The main area has a title 'default-allow-ssh' with a shield icon and 'Avalanche-nsg' underneath. There are three tabs at the top: 'Save' (highlighted in yellow), 'Discard', and 'Basic'. The 'Inbound rules' section shows one rule: '1000: default-allow-ssh' with 'Any' selected. The 'Source' dropdown is set to 'IP Addresses' and contains '143.24.12.123'. The 'Source port ranges' field is set to '\*'. The 'Destination' field is set to 'Any'. The 'Destination port ranges' field is set to '22'. Under 'Protocol', 'TCP' is selected. The 'Action' button is set to 'Allow'. The 'Priority' field is set to '100'. At the bottom, the 'Name' field is set to 'default-allow-ssh' and the 'Description' field is empty.

Then select "+ Add an inbound rule" to add another rule for RPC access, this should also be restricted to only your IP. Change Source to "IP Addresses" and enter in your public IP returned from google into the Source IP field. This time change the "Destination port ranges" field to 9650 and select "TCP" as the protocol. Change the priority to 110 and give it a name of "Avalanche\_RPC" and press Add.

## Create network security group

Name \*

Avalanche-nsg

Inbound rules ⓘ

|                        |                                     |
|------------------------|-------------------------------------|
| 100: default-allow-ssh | <input checked="" type="checkbox"/> |
| 143.200.0.0/16         | <input type="checkbox"/>            |
| SSH (TCP/22)           | <input type="checkbox"/>            |

< >

+ Add an inbound rule

Outbound rules ⓘ

|                        |                          |
|------------------------|--------------------------|
| No results             | <input type="checkbox"/> |
| + Add an outbound rule | <input type="checkbox"/> |

**Add inbound security rule** Avalanche-nsg

Basic

Source \* ⓘ

IP Addresses

Source IP addresses/CIDR ranges \* ⓘ

143.200.0.0/16

Source port ranges \* ⓘ

\*

Destination \* ⓘ

Any

Destination port ranges \* ⓘ

9650

Protocol \*

Any TCP UDP ICMP

Action \*

Allow Deny

Priority \* ⓘ

110

Name \*

Avalanche\_RPC

Description

OK Add

Select "+ Add an inbound rule" to add a final rule for the Avalanche Protocol so that other nodes can communicate with your node. This rule needs to be open to everyone so keep "Source" set to "Any." Change the Destination port range to "9651" and change the protocol to "TCP." Enter a priority of 120 and a name of Avalanche\_Proto and press Add.

Home > New > Ubuntu Server 18.04 LTS > Create a network security group

## Create network security group

Name \*

Inbound rules ⓘ

|                                 |                       |
|---------------------------------|-----------------------|
| 100: default-allow-ssh          | ✓                     |
| 143.20.10.10: SSH (TCP/22)      | ✓                     |
| 110: Avalanche_RPC              | ✓                     |
| 143.20.10.10: Custom (TCP/9650) | ✓                     |
| <   >                           | + Add an inbound rule |

Outbound rules ⓘ

No results  
+ Add an outbound rule

## Add inbound security rule

Avalanche-nsg

Basic

Source \* ⓘ

Source port ranges \* ⓘ

Destination \* ⓘ

Destination port ranges \* ⓘ

Protocol \*

Any TCP UDP ICMP

Action \*

Allow Deny

Priority \*

Name \*

Description

Add

OK

The network security group should look like the below (albeit your public IP address will be different) and press OK.

## Create network security group

Name \*

Avalanche-nsg

Inbound rules ⓘ

|                                    |   |
|------------------------------------|---|
| 100: default-allow-ssh             | ✓ |
| 143.2 [REDACTED] SSH (TCP/22)      | ✓ |
| 110: Avalanche_RPC                 | ✓ |
| 143.2 [REDACTED] Custom (TCP/9650) | ✓ |
| 120: Avalanche_Protocol            | ✓ |
| Any                                | ✓ |
| Custom (TCP/9651)                  | ✓ |

< > + Add an inbound rule

Outbound rules ⓘ

|            |
|------------|
| No results |
| < >        |

+ Add an outbound rule

OK

Leave the other settings as default and then press "Review + create" to create the Virtual machine.

## Create a virtual machine

Virtual network \* ⓘ

(new) Avalanche\_group-vnet

Create new

Subnet \* ⓘ

(new) default (10.0.0.0/24)

Public IP ⓘ

(new) Avalanche-ip

Create new

NIC network security group ⓘ

None  Basic  Advanced

Configure network security group \*

(new) Avalanche-nsg

Create new

Accelerated networking ⓘ

On  Off

The selected VM size does not support accelerated networking.

### Load balancing

You can place this virtual machine in the backend pool of an existing Azure load balancing solution. [Learn more](#)

Place this virtual machine behind an existing load balancing solution?

Yes  No

Review + create

< Previous

Next : Management >

First it will perform a validation test. If you receive an error here, make sure you selected Pay-As-You-Go subscription model and you are not using the Free Trial subscription as Spot instances are not available. Verify everything looks correct and press "Create"

## Create a virtual machine

Validation passed

Basics Disks Networking Management Advanced Tags Review + create

### PRODUCT DETAILS

Standard D2s\_v4  
by Microsoft  
[Terms of use](#) | [Privacy policy](#)

Subscription credits apply ⓘ

**0.0129 USD/hr**

[Pricing for other VM sizes](#)

### TERMS

By clicking "Create", I (a) agree to the legal terms and privacy statement(s) associated with the Marketplace offering(s) listed above; (b) authorize Microsoft to bill my current payment method for the fees associated with the offering(s), with the same billing frequency as my Azure subscription; and (c) agree that Microsoft may share my contact, usage and transactional information with the provider(s) of the offering(s) for support, billing and other transactional activities. Microsoft does not provide rights for third-party offerings. See the [Azure Marketplace Terms](#) for additional details.

### Basics

|                      |                                         |
|----------------------|-----------------------------------------|
| Subscription         | Pay-As-You-Go                           |
| Resource group       | (new) Avalanche_group                   |
| Virtual machine name | Avalanche                               |
| Region               | North Europe                            |
| Availability options | No infrastructure redundancy required   |
| Image                | Ubuntu Server 18.04 LTS - Gen1          |
| Size                 | Standard D2s_v4 (2 vcpus, 8 GiB memory) |
| Authentication type  | SSH public key                          |
| Username             | ubuntu                                  |
| Key pair name        | Avalanche_key                           |
| Azure Spot           | Yes (Stop / Deallocate)                 |
| Azure Spot max price | Capacity                                |

### Disks

|                       |             |
|-----------------------|-------------|
| OS disk type          | Premium SSD |
| Use managed disks     | Yes         |
| Use ephemeral OS disk | No          |

### Networking

|                 |                             |
|-----------------|-----------------------------|
| Virtual network | (new) Avalanche_group-vnet  |
| Subnet          | (new) default (10.0.0.0/24) |

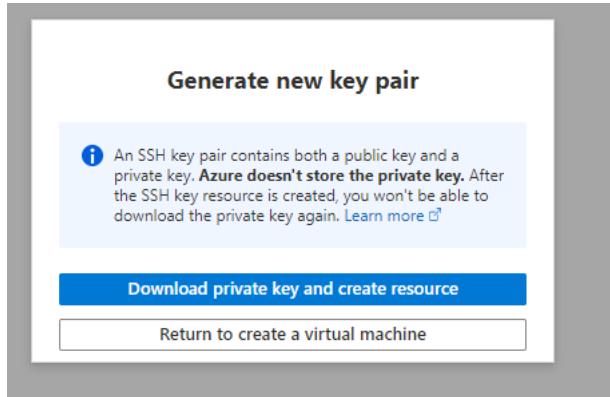
**Create**

< Previous

Next >

[Download a template for automation](#)

You should then receive a prompt asking you to generate a new key pair to connect your virtual machine. Select "Download private key and create resource" to download the private key to your PC.



Once your deployment has finished, select "Go to resource"

## ✓ Your deployment is complete

Deployment name: CreateVm-Canonical.UbuntuServer-18.04-LTS-2... Start time: 11/16/20  
Subscription: Pay-As-You-Go Correlation ID: 8422  
Resource group: Avalanche\_group

∨ Deployment details ([Download](#))

^ Next steps

[Setup auto-shutdown](#) Recommended  
[Monitor VM health, performance and network dependencies](#) Recommended  
[Run a script inside the virtual machine](#) Recommended

[Go to resource](#)

[Create another VM](#)

## Change the Provisioned Disk Size

By default, the Ubuntu VM will be provisioned with a 30 GB Premium SSD. You should increase this to 250 GB, to allow for database growth.

**Avalanche** ⚡

Virtual machine

Search (Ctrl+ /)

Connect Start Restart Stop Capture Delete Refresh Open

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Settings

Networking

Connect

Disks

Stop this virtual machine

Do you want to stop 'Avalanche'?

**Yes** **No**

Subscription (change) : Pay-As-You-Go

Subscription ID : debecfbb-7622-437a-a680-bc2db87704c6

Tags (change) : [Click here to add tags](#)

Properties Monitoring Capabilities (8) Recommendations Tutorials

To change the Disk size, the VM needs to be stopped and deallocated. Select "Stop" and wait for the status to show deallocated. Then select "Disks" on the left.

**Avalanche** Virtual machine

Search (Ctrl+ /) Connect Start Restart Stop Capture Delete Refresh Open in mobile

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Networking Connect Disks Size Schedules

**Essentials**

|                                                        |       |
|--------------------------------------------------------|-------|
| Resource group (change) : Avalanche_group              | Ope   |
| Status : Stopped (deallocated)                         | Size  |
| Location : North Europe                                | Publ  |
| Subscription (change) : Pay-As-You-Go                  | Virtu |
| Subscription ID : debecfbb-7622-437a-a680-bc2db87704c6 | DNS   |
| Tags (change) : Click here to add tags                 |       |

Properties Monitoring Capabilities (8) Recommendations Tutorials

**Virtual machine**

|                  |           |
|------------------|-----------|
| Computer name    | Avalanche |
| Operating system | Linux     |

Select the Disk name that's current provisioned to modify it

**Avalanche | Disks** Virtual machine

Search (Ctrl+ /) Save Discard Refresh Additional settings

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Settings Networking Connect

**OS disk**

Swap OS disk

|                                       |              |
|---------------------------------------|--------------|
| Disk name                             | Storage type |
| Avalanche_OsDisk_1_06bab1099b7f47bcbb | Premium SSD  |

**Data disks**

Filter by name

Showing 0 of 0 attached data disks

Select "Size + performance" on the left under settings and change the size to 250 GB and press "Resize"

Home > CreateVm-Canonical.UbuntuServer-18.04-LTS-20201116133722 > Avalanche > Avalanche\_OsDisk\_1\_06bab1099b7f47bcbbec

## Avalanche\_OsDisk\_1\_06bab1099b7f47bcbbec7318432d3cad | Size + performance

Disk

Search (Ctrl+ /) <<

| Disk SKU ⓘ  |           |                  |   |
|-------------|-----------|------------------|---|
| Premium SSD | ▼         |                  |   |
| Size        | Disk tier | Provisioned IOPS | ⋮ |
| 4 GiB       | P1        | 120              | ⋮ |
| 8 GiB       | P2        | 120              | ⋮ |
| 16 GiB      | P3        | 120              | ⋮ |
| 32 GiB      | P4        | 120              | ⋮ |
| 64 GiB      | P6        | 240              | ⋮ |
| 128 GiB     | P10       | 500              | ⋮ |
| 256 GiB     | P15       | 1100             | ⋮ |
| 512 GiB     | P20       | 2300             | ⋮ |
| 1024 GiB    | P30       | 5000             | ⋮ |
| 2048 GiB    | P40       | 7500             | ⋮ |
| 4096 GiB    | P50       | 7500             | ⋮ |
| 8192 GiB    | P60       | 16000            | ⋮ |
| 16384 GiB   | P70       | 18000            | ⋮ |
| 32767 GiB   | P80       | 20000            | ⋮ |

Encryption

Networking

Disk Export

Properties

Locks

Monitoring

Metrics

Automation

Tasks

Export template

Support + troubleshooting

New support request

Custom disk size (GiB) \* ⓘ

64 ✓

**Resize** Discard

Doing this now will also extend the partition automatically within Ubuntu. To go back to the virtual machine overview page, select Avalanche in the navigation setting.

Home > CreateVm-Canonical.UbuntuServer-18.04-LTS-20201116133722 > Avalanche

## Avalanche | Disks

Virtual machine

Search (Ctrl+ /) <<

Save Discard Refresh Additional settings

OS disk

Swap OS disk

| Disk name                               | Storage type | Size (GiB) | Max IOPS |
|-----------------------------------------|--------------|------------|----------|
| Avalanche_OsDisk_1_06bab1099b7f47bcbbec | Premium SSD  | 64         | 240      |

Data disks

Filter by name

Then start the VM

The screenshot shows the Azure portal interface for a virtual machine named 'Avalanche'. The top navigation bar includes 'Home', 'CreateVm-Canonical.UbuntuServer-18.04-LTS-20201116133722', and a search bar. Below the navigation is a toolbar with icons for 'Connect', 'Start' (highlighted in yellow), 'Restart', 'Stop', 'Capture', 'Delete', 'Refresh', and 'Open in monitor'. A sidebar on the left lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings', and 'Networking'. The main content area is titled 'Essentials' and displays the following information:

- Resource group (change) : Avalanche\_group
- Status : Stopped (deallocated)
- Location : North Europe
- Subscription (change) : Pay-As-You-Go
- Subscription ID : debecfbb-7622-437a-a680-bc2db87704c6
- Tags (change) : Click here to add tags

## Connect to the Avalanche Node

The following instructions show how to connect to the Virtual Machine from a Windows 10 machine. For instructions on how to connect from a Ubuntu machine see the [AWS tutorial](#).

On your local PC, create a folder on the root of the C: drive called Avalanche and then move the Avalanche\_key.pem file you downloaded before into the folder. Then right click the file and select Properties. Go to the security tab and select "Advanced" at the bottom

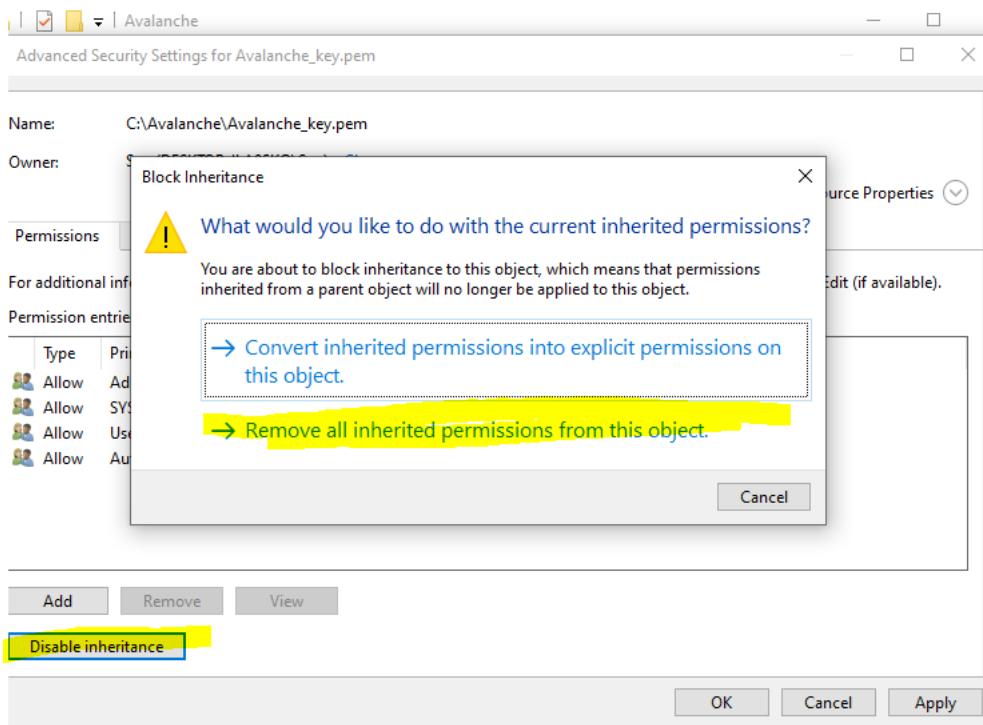
The screenshot shows a Windows File Explorer window with a folder structure on the left and a file named 'Avalanche\_key.pem' selected on the right. A properties dialog box is open over the file, specifically the 'Security' tab. The dialog shows the following details:

- Object name: C:\Avalanche\Avalanche\_key.pem
- Group or user names:
  - Authenticated Users
  - SYSTEM
  - Administrators (DESKTOP-ILA95KQ\Administrators)
  - Users (DESKTOP-ILA95KQ\Users)
- To change permissions, click Edit... (button highlighted in yellow)
- Permissions for Authenticated Users:

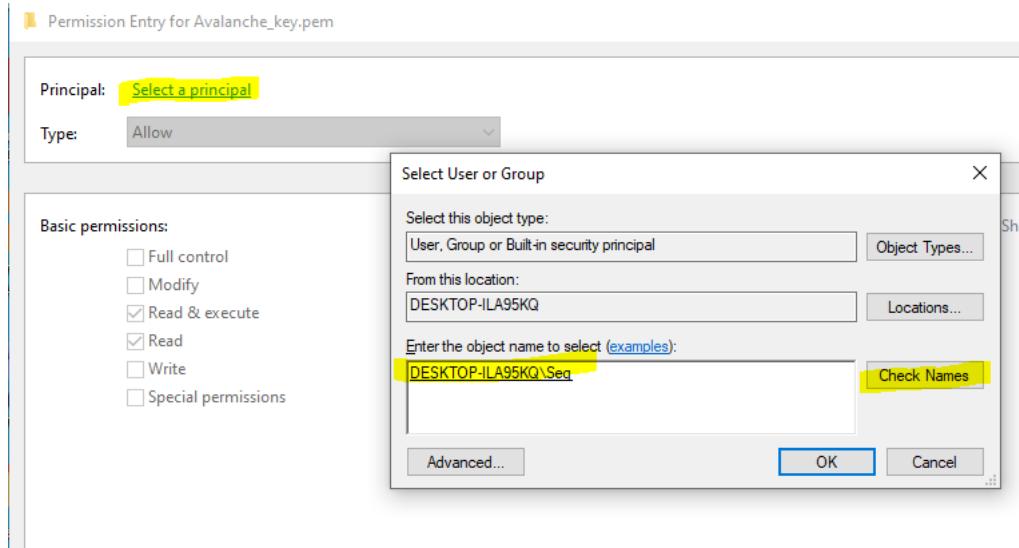
|                     | Allow | Deny |
|---------------------|-------|------|
| Full control        |       |      |
| Modify              | ✓     |      |
| Read & execute      | ✓     |      |
| Read                | ✓     |      |
| Write               | ✓     |      |
| Special permissions |       |      |
- For special permissions or advanced settings, click Advanced... (button highlighted in yellow)

At the bottom of the dialog are 'OK', 'Cancel', and 'Apply' buttons.

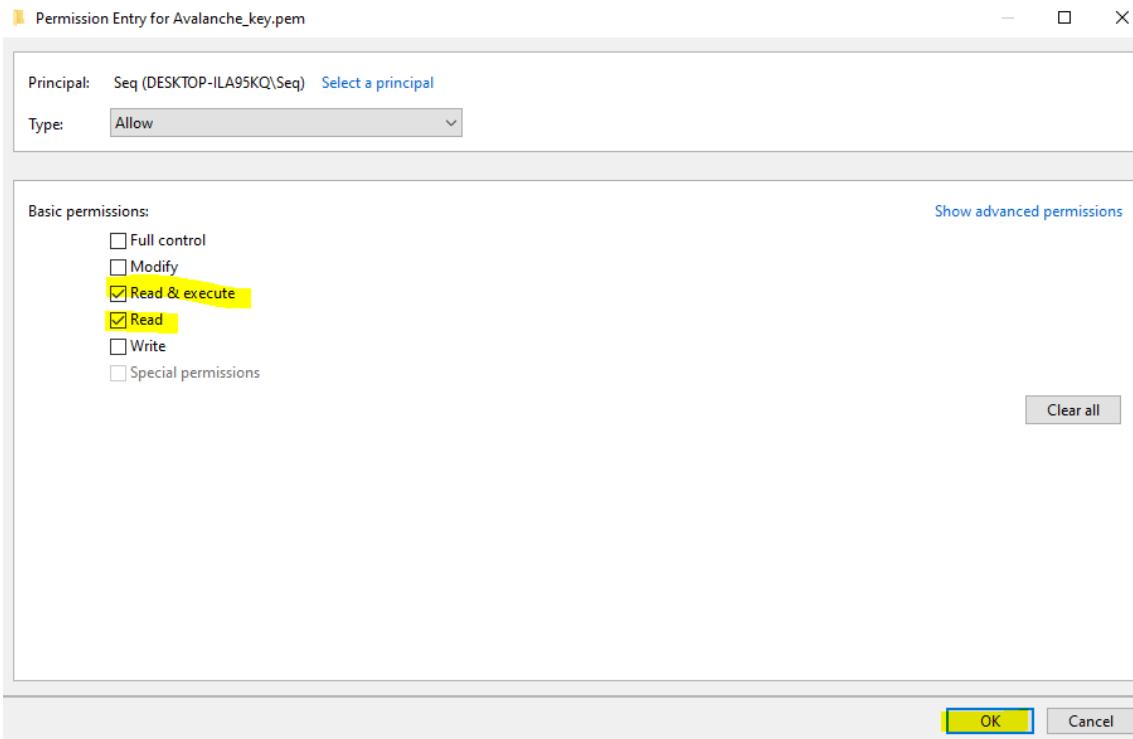
Select "Disable inheritance" and then "Remove all inherited permissions from this object" to remove all existing permissions on that file.



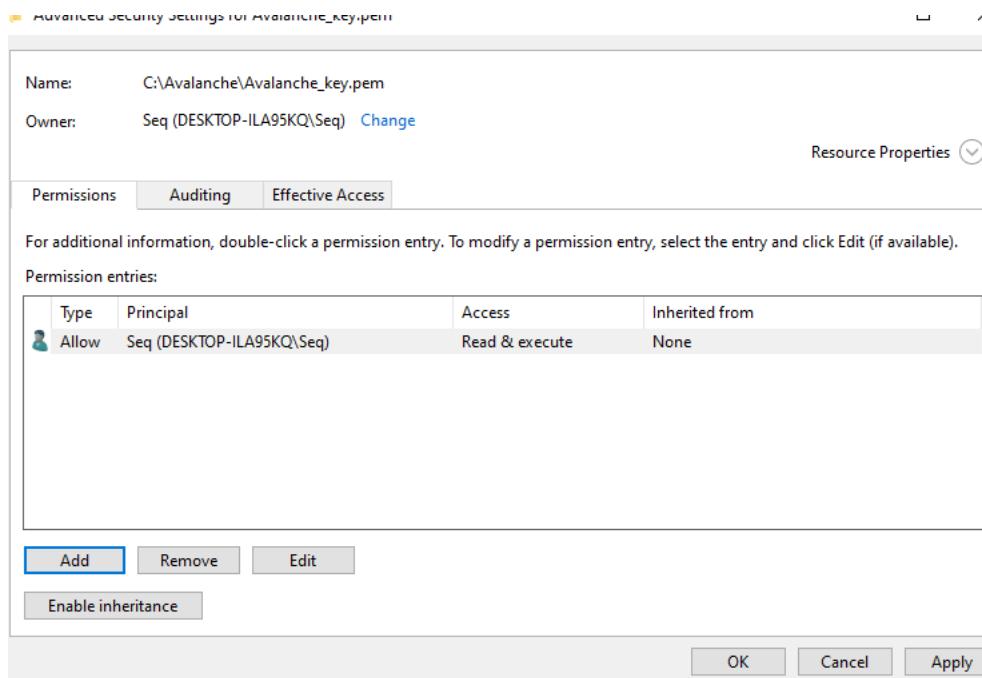
Then select "Add" to add a new permission and choose "Select a principal" at the top. From the pop-up box enter in your user account that you use to log into your machine. In this example I log on with a local user called Seq, you may have a Microsoft account that you use to log in, so use whatever account you login to your PC with and press "Check Names" and it should underline it to verify and press OK.



Then from the permissions section make sure only "Read & Execute" and "Read" are selected and press OK.



It should look something like the below, except with a different PC name / user account. This just means the key file can't be modified or accessed by any other accounts on this machine for security purposes so they can't access your Avalanche Node.

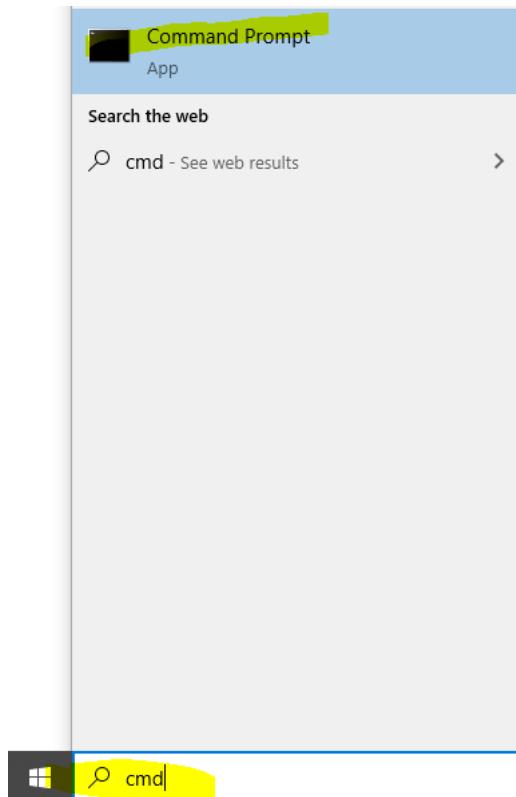


#### Find your Avalanche Node Public IP

From the Azure Portal make a note of your static public IP address that has been assigned to your node.

| Essentials                                             |                                                       |
|--------------------------------------------------------|-------------------------------------------------------|
| Resource group (change) : Avalanche_group              | Operating system : Linux (ubuntu 18.04)               |
| Status : Running                                       | Size : Standard D2s_v4 (2 vcpus, 8 GiB memory)        |
| Location : North Europe                                | Public IP address : 13.74.10.81                       |
| Subscription (change) : Pay-As-You-Go                  | Virtual network/subnet : Avalanche_group-vnet/default |
| Subscription ID : debecfbb-7622-437a-a680-bc2db87704c6 | DNS name : Configure                                  |
| Tags (change) : Click here to add tags                 |                                                       |

To log onto the Avalanche node, open command prompt by searching for `cmd` and selecting "Command Prompt" on your Windows 10 machine.



Then use the following command and replace EnterYourAzureIPHere with the static IP address shown on the Azure portal.

```
ssh -i C:\Avalanche\Avalanche_key.pem ubuntu@EnterYourAzureIPHere
```

for my example its:

```
ssh -i C:\Avalanche\Avalanche_key.pem ubuntu@13.74.10.81
```

The first time you connect you will receive a prompt asking to continue, enter yes.

```
c:\>ssh -i C:\Avalanche\Avalanche_key.pem ubuntu@13.74.10.81
The authenticity of host '13.74.10.81 (13.74.10.81)' can't be established.
ECDSA key fingerprint is SHA256:mrmHzKqG13epwfx2ckewtXmaafK6CVUdADhnRduKy4.
Are you sure you want to continue connecting (yes/no)? yes
```

You should now be connected to your Node.

```
c:\>ssh -i C:\Avalanche\Avalanche_key.pem ubuntu@13.74.10.81
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.4.0-1031-azure x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

System information as of Mon Nov 16 15:15:53 UTC 2020

System load: 0.0 Processes: 113Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.4.0-1031-azure x86_64)
* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage

System information as of Mon Nov 16 15:15:53 UTC 2020

System load: 0.0 Processes: 113
Usage of /: 2.1% of 61.86GB Users logged in: 0
Memory usage: 2% IP address for eth0: 10.0.0.4
Swap usage: 0%

0 packages can be updated.
0 updates are security updates.

New release '20.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Mon Nov 16 15:15:31 2020 from 82.70.49.154
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@Avalanche:~$
```

The following section is taken from Colin's excellent tutorial for [configuring an Avalanche Node on Amazon's AWS](#).

### Update Linux with Security Patches

Now that we are on our node, it's a good idea to update it to the latest packages. To do this, run the following commands, one-at-a-time, in order:

```
sudo apt update
sudo apt upgrade -y
sudo reboot
```

```

ubuntu@Avalanche:~$ sudo apt update
Hit:1 http://azure.archive.ubuntu.com/ubuntu bionic InRelease
Get:2 http://azure.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Get:3 http://azure.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:4 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Get:5 http://azure.archive.ubuntu.com/ubuntu bionic/universe amd64 Packages [8570 kB]
Get:6 http://azure.archive.ubuntu.com/ubuntu bionic/universe Translation-en [4941 kB]
Get:7 http://azure.archive.ubuntu.com/ubuntu bionic/multiverse amd64 Packages [151 kB]
Get:8 http://azure.archive.ubuntu.com/ubuntu bionic/multiverse Translation-en [108 kB]
Get:9 http://azure.archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [1753 kB]
Get:10 http://azure.archive.ubuntu.com/ubuntu bionic-updates/main Translation-en [371 kB]
Get:11 http://azure.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages [1692 kB]
Get:12 http://azure.archive.ubuntu.com/ubuntu bionic-updates/universe Translation-en [355 kB]
Get:13 http://azure.archive.ubuntu.com/ubuntu bionic-updates/multiverse amd64 Packages [32.3 kB]
Get:14 http://azure.archive.ubuntu.com/ubuntu bionic-updates/multiverse Translation-en [7012 B]
Get:15 http://azure.archive.ubuntu.com/ubuntu bionic-backports/main amd64 Packages [10.0 kB]
Get:16 http://azure.archive.ubuntu.com/ubuntu bionic-backports/main Translation-en [4764 B]
Get:17 http://azure.archive.ubuntu.com/ubuntu bionic-backports/universe amd64 Packages [10.3 kB]
Get:18 http://azure.archive.ubuntu.com/ubuntu bionic-backports/universe Translation-en [4588 B]
Get:19 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [1425 kB]
Get:20 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [1088 kB]
Get:21 http://security.ubuntu.com/ubuntu bionic-security/universe Translation-en [243 kB]
Get:22 http://security.ubuntu.com/ubuntu bionic-security/multiverse amd64 Packages [13.3 kB]
Get:23 http://security.ubuntu.com/ubuntu bionic-security/multiverse Translation-en [2996 B]
Fetched 21.0 MB in 3s (6632 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
All packages are up to date.
ubuntu@Avalanche:~$ sudo apt upgrade -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages were automatically installed and are no longer required:
 grub-pc-bin linux-headers-4.15.0-123
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
ubuntu@Avalanche:~$ sudo reboot

```

This will make our instance up to date with the latest security patches for our operating system. This will also reboot the node. We'll give the node a minute or two to boot back up, then log in again, same as before.

### Set up the Avalanche Node

Now we'll need to set up our Avalanche node. To do this, follow the [Set Up Avalanche Node With Installer](#) tutorial which automates the installation process. You will need the "IPv4 Public IP" copied from the Azure Portal we set up earlier.

Once the installation is complete, our node should now be bootstrapping! We can run the following command to take a peek at the latest status of the AvalancheGo node:

```
sudo systemctl status avalanchego
```

To check the status of the bootstrap, we'll need to make a request to the local RPC using `curl`. This request is as follows:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.isBootstrapped",
 "params": {
 "chain": "X"
 }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

The node can take some time (upward of an hour at this moment writing) to bootstrap. Bootstrapping means that the node downloads and verifies the history of the chains. Give this some time. Once the node is finished bootstrapping, the response will be:

```
{
 "jsonrpc": "2.0",
 "result": {
 "isBootstrapped": true
 },
 "id": 1
}
```

We can always use `sudo systemctl status avalanchego` to peek at the latest status of our service as before, as well.

## Get Your NodeID

We absolutely must get our NodeID if we plan to do any validating on this node. This is retrieved from the RPC as well. We call the following curl command to get our NodeID.

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

If all is well, the response should look something like:

```
{"jsonrpc": "2.0", "result": {"nodeID": "NodeID-Lve2PzuCvXZrqn8Stqwy9vWZux6VyGUCR"}, "id": 1}
```

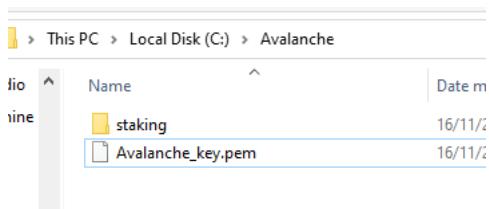
That portion that says, "NodeID-Lve2PzuCvXZrqn8Stqwy9vWZux6VyGUCR" is our NodeID, the entire thing. Copy that and keep that in our notes. There's nothing confidential or secure about this value, but it's an absolute must for when we submit this node to be a validator.

## Backup Your Staking Keys

The last thing that should be done is backing up our staking keys in the untimely event that our instance is corrupted or terminated. It's just good practice for us to keep these keys. To back them up, we use the following command:

```
scp -i C:\Avalanche\avalanche_key.pem -r ubuntu@EnterYourAzureIPHere:/home/ubuntu/.avalanchego/staking C:\Avalanche
```

As before, we'll need to replace "EnterYourAzureIPHere" with the appropriate value that we retrieved. This backs up our staking key and staking certificate into the C:\Avalanche folder we created before.



**sidebar\_position: 7 description: Detailed instructions for running an Avalanche node with Alibaba Cloud**

## Run an Avalanche Node with Alibaba Cloud

Detailed instructions can be found [here](#).

**sidebar\_position: 8 description: Detailed instructions for running an Avalanche node with Tencent Cloud**

## Run an Avalanche Node with Tencent Cloud

Detailed instructions can be found [here](#).

**sidebar\_position: 1 description: Detailed instructions for running an Avalanche node using the install script.**

## Run an Avalanche Node Using the Install Script

We have a shell (bash) script that installs AvalancheGo on your computer. This script sets up full, running node in a matter of minutes with minimal user input required. Script can also be used for unattended, [automated installs](#).

### Before You Start

Avalanche is an incredibly lightweight protocol, so nodes can run on commodity hardware with the following minimum specifications.

- CPU: Equivalent of 8 AWS vCPU
- RAM: 16 GiB
- Storage: 1 TiB
- OS: Ubuntu 20.04 or MacOS >= 12
- Network: sustained 5Mbps up/down bandwidth

:::note

HW requirements shall scale with the amount of AVAX staked on the node and/or network activity. Nodes with big stakes (100k+ AVAX) will need more powerful machines than listed, and will use more bandwidth as well.

:::

This install script assumes:

- AvalancheGo is not running and not already installed as a service
- User running the script has superuser privileges (can run `sudo`)

## Environment Considerations

If you run a different flavor of Linux, the script might not work as intended. It assumes `systemd` is used to run system services. Other Linux flavors might use something else, or might have files in different places than is assumed by the script. It will probably work on any distribution that uses `systemd` but it has been developed for and tested on Ubuntu.

If you have a node already running on the computer, stop it before running the script. Script won't touch the node working directory so you won't need to bootstrap the node again.

### Node Running from Terminal

If your node is running in a terminal stop it by pressing `ctrl+C`.

### Node Running as a Service

If your node is already running as a service, then you probably don't need this script. You're good to go.

### Node Running in the Background

If your node is running in the background (by running with `nohup`, for example) then find the process running the node by running `ps aux | grep avalanche`. This will produce output like:

```
ubuntu 6834 0.0 0.0 2828 676 pts/1 S+ 19:54 0:00 grep avalanche
ubuntu 2630 26.1 9.4 2459236 753316 ? S1 Dec02 1220:52 /home/ubuntu/build/avalanchego
```

Look for line that doesn't have `grep` on it. In this example, that is the second line. It shows information about your node. Note the process id, in this case, `2630`. Stop the node by running `kill -2 2630`.

## Node Working Files

If you previously ran an AvalancheGo node on this computer, you will have local node files stored in `$HOME/.avalanchego` directory. Those files will not be disturbed, and node set up by the script will continue operation with the same identity and state it had before. That being said, for your node's security, back up `staker.crt` and `staker.key` files, found in `$HOME/.avalanchego/staking` and store them somewhere secure. You can use those files to recreate your node on a different computer if you ever need to. Check out this [tutorial](#) for backup and restore procedure.

## Networking Considerations

To run successfully, AvalancheGo needs to accept connections from the Internet on the network port `9651`. Before you proceed with the installation, you need to determine the networking environment your node will run in.

### Running on a Cloud Provider

If your node is running on a cloud provider computer instance, it will have a static IP. Find out what that static IP is, or set it up if you didn't already. The script will try to find out the IP by itself, but that might not work in all environments, so you will need to check the IP or enter it yourself.

### Running on a Home Connection

If you're running a node on a computer that is on a residential internet connection, you have a dynamic IP; that is, your IP will change periodically. The install script will configure the node appropriately for that situation. But, for a home connection, you will need to set up inbound port forwarding of port `9651` from the internet to the computer the node is installed on.

As there are too many models and router configurations, we cannot provide instructions on what exactly to do, but there are online guides to be found (like [this](#), or [this](#)), and your service provider support might help too.

:::warning

Please note that a fully connected Avalanche node maintains and communicates over a couple of thousand of live TCP connections. For some low-powered and older home routers that might be too much to handle. If that is the case you may experience lagging on other computers connected to the same router, node getting benched, failing to sync and similar issues.

:::

## Running the Script

So, now that you prepared your system and have the info ready, let's get to it.

To download and run the script, enter the following in the terminal:

```
wget -nd -m https://raw.githubusercontent.com/ava-labs/avalanche-docs/master/scripts/avalanchego-installer.sh; \
chmod 755 avalanchego-installer.sh; \
./avalanchego-installer.sh
```

And we're off! The output should look something like this:

```
AvalancheGo installer

```

```

Preparing environment...
Found arm64 architecture...
Looking for the latest arm64 build...
Will attempt to download:
 https://github.com/ava-labs/avalanchego/releases/download/v1.1.1/avalanchego-linux-arm64-v1.1.1.tar.gz
avalanchego-linux-arm64-v1.1.1.tar.gz 100%[=====] 29.83M
75.8MB/s in 0.4s
2020-12-28 14:57:47 URL:https://github-production-release-asset-2e65be.s3.amazonaws.com/246387644/f4d27b00-4161-11eb-8fb2-
156a992fd2c8?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNYAX4CSVEH53A%2F20201228%2Fus-east-
1%2F%32Faws4_request&X-Amz-Date=20201228T145747Z&X-Amz-Expires=300&X-Amz-
Signature=ea838877f39ae940a37a076137c4c2689494c7e683cb95a5a4714c062e6ba018&X-Amz-
SignedHeaders=host&actor_id=0&key_id=0&repo_id=246387644&response-content-disposition=attachment%3B%20filename%3Davalanchego-
linux-arm64-v1.1.1.tar.gz&response-content-type=application%2Foctet-stream [31283052/31283052] -> "avalanchego-linux-arm64-
v1.1.1.tar.gz" [1]
Unpacking node files...
avalanchego-v1.1.1/plugins/
avalanchego-v1.1.1/plugins/evm
avalanchego-v1.1.1/avalanchego
Node files unpacked into /home/ubuntu/avalanche-node

```

And then the script will prompt you for information about the network environment:

```

To complete the setup some networking information is needed.
Where is the node installed:
1) residential network (dynamic IP)
2) cloud provider (static IP)
Enter your connection type [1,2]:

```

enter 1 if you have dynamic IP, and 2 if you have a static IP. If you are on a static IP, it will try to auto-detect the IP and ask for confirmation.

```
Detected '3.15.152.14' as your public IP. Is this correct? [y,n]:
```

Confirm with y , or n if the detected IP is wrong (or empty), and then enter the correct IP at the next prompt.

Next, you have to set up RPC port access for your node. Those are used to query the node for its internal state, to send commands to the node, or to interact with the platform and its chains (sending transactions, for example). You will be prompted:

```
RPC port should be public (this is a public API node) or private (this is a validator)? [public, private]:
```

- private : this setting only allows RPC requests from the node machine itself.
- public : this setting exposes the RPC port to all network interfaces.

As this is a sensitive setting you will be asked to confirm if choosing public . Please read the following note carefully:

:::note

If you choose to allow RPC requests on any network interface you will need to set up a firewall to only let through RPC requests from known IP addresses, otherwise your node will be accessible to anyone and might be overwhelmed by RPC calls from malicious actors! If you do not plan to use your node to send RPC calls remotely, enter private .

:::

The script will then prompt you to choose whether to enable state sync setting or not:

```
Do you want state sync bootstrapping to be turned on or off? [on, off]:
```

Turning state sync on will greatly increase the speed of bootstrapping, but will sync only the current network state. If you intend to use your node for accessing historical data (archival node) you should select off . Otherwise, select on . Validators can be bootstrapped with state sync turned on.

The script will then continue with system service creation and finish with starting the service:

```
Created symlink /etc/systemd/system/multi-user.target.wants/avalanchego.service → /etc/systemd/system/avalanchego.service.
```

```
Done!
```

```
Your node should now be bootstrapping.
Node configuration file is /home/ubuntu/.avalanchego/configs/node.json
C-Chain configuration file is /home/ubuntu/.avalanchego/configs/chains/C/config.json
Plugin directory, for storing subnet VM binaries, is /home/ubuntu/.avalanchego/plugins
To check that the service is running use the following command (q to exit):
sudo systemctl status avalanchego
To follow the log use (ctrl-c to stop):
sudo journalctl -u avalanchego -f
```

```
Reach us over on https://chat.avax.network if you're having problems.
```

The script is finished, and you should see the system prompt again.

## Post Installation

AvalancheGo should be running in the background as a service. You can check that it's running with:

```
sudo systemctl status avalanchego
```

This will print the node's latest logs, which should look like this:

```
● avalanchego.service - AvalancheGo systemd service
 Loaded: loaded (/etc/systemd/system/avalanchego.service; enabled; vendor preset: enabled)
 Active: active (running) since Tue 2021-01-05 10:38:21 UTC; 51s ago
 Main PID: 2142 (avalanchego)
 Tasks: 8 (limit: 4495)
 Memory: 223.0M
 CGroup: /system.slice/avalanchego.service
 └─2142 /home/ubuntu/avalanche-node/avalanchego --public-ip-resolution-service=opendns --http-host=

Jan 05 10:38:45 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:45] <P Chain> avalanchego/vms/platformvm/vm.go#322: initializing last accepted block as 2FUFVVxPxbTpKNN39mcGSzsmGroYES4NZRdw3mJgNvMkMiMHJ9e
Jan 05 10:38:45 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:45] <P Chain> avalanchego/snow/engine/snowman/transitive.go#58: initializing consensus engine
Jan 05 10:38:45 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:45] avalanchego/api/server.go#143: adding route /ext/bc/1111111111111111111111111111LpoYY
Jan 05 10:38:45 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:45] avalanchego/api/server.go#88: HTTP API server listening on ":9650"
Jan 05 10:38:58 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:58] <P Chain> avalanchego/snow/engine/common/bootstrapper.go#185: Bootstrapping started syncing with 1 vertices in the accepted frontier
Jan 05 10:39:02 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:39:02] <P Chain> avalanchego/snow/engine/snowman/bootstrap/bootstrapper.go#210: fetched 2500 blocks
Jan 05 10:39:04 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:39:04] <P Chain> avalanchego/snow/engine/snowman/bootstrap/bootstrapper.go#210: fetched 5000 blocks
Jan 05 10:39:06 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:39:06] <P Chain> avalanchego/snow/engine/snowman/bootstrap/bootstrapper.go#210: fetched 7500 blocks
Jan 05 10:39:09 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:39:09] <P Chain> avalanchego/snow/engine/snowman/bootstrap/bootstrapper.go#210: fetched 10000 blocks
Jan 05 10:39:11 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:39:11] <P Chain> avalanchego/snow/engine/snowman/bootstrap/bootstrapper.go#210: fetched 12500 blocks
```

Note the `active (running)` which indicates the service is running OK. You may need to press `q` to return to the command prompt.

To find out your NodeID, which is used to identify your node to the network, run the following command:

```
sudo journalctl -u avalanchego | grep "NodeID"
```

It will produce output like:

```
Jan 05 10:38:38 ip-172-31-30-64 avalanchego[2142]: INFO [01-05|10:38:38] avalanchego/node/node.go#428: Set node's ID to 6seStrauyCnVV7NEVwRbfaT9B6EnXEzfY
```

Prepend `NodeID-` to the value to get, for example, `NodeID-6seStrauyCnVV7NEVwRbfaT9B6EnXEzfY`. Store that; it will be needed for staking or looking up your node.

Your node should be in the process of bootstrapping now. You can monitor the progress by issuing the following command:

```
sudo journalctl -u avalanchego -f
```

Press `ctrl+C` when you wish to stop reading node output.

## Stopping the Node

To stop AvalancheGo, run:

```
sudo systemctl stop avalanchego
```

To start it again, run:

```
sudo systemctl start avalanchego
```

## Node Upgrade

AvalancheGo is an ongoing project and there are regular version upgrades. Most upgrades are recommended but not required. Advance notice will be given for upgrades that are not backwards compatible. When a new version of the node is released, you will notice log lines like:

```
Jan 08 10:26:45 ip-172-31-16-229 avalanchego[6335]: INFO [01-08|10:26:45] avalanchego/network/peer.go#526: beacon
9CkG9MBNavnw7EVSRsuFr7ws9gascDQy3 attempting to connect with newer version avalanche/1.1.1. You may want to update your client
```

It is recommended to always upgrade to the latest version, because new versions bring bug fixes, new features and upgrades.

To upgrade your node, just run the installer script again:

```
./avalanchego-installer.sh
```

It will detect that you already have AvalancheGo installed:

```
AvalancheGo installer

Preparing environment...
Found 64bit Intel/AMD architecture...
Found AvalancheGo systemd service already installed, switching to upgrade mode.
Stopping service...
```

It will then upgrade your node to the latest version, and after it's done, start the node back up, and print out the information about the latest version:

```
Node upgraded, starting service...
New node version:
avalanche/1.1.1 [network=mainnet, database=v1.0.0, commit=f76f1fd5f99736cf468413bbac158d6626f712d2]
Done!
```

## Advanced Node Configuration

Without any additional arguments, the script installs the node in a most common configuration. But the script also enables various advanced options to be configured, via the command line prompts. Following is a list of advanced options and their usage:

- `admin` - [Admin API](#) will be enabled
- `archival` - disables database pruning and preserves the complete transaction history
- `state-sync` - if `on` state-sync for the C-Chain is used, if `off` it will use regular transaction replay to bootstrap; state-sync is much faster, but has no historical data
- `db-dir` - use to provide the full path to the location where the database will be stored
- `fuji` - node will connect to Fuji testnet instead of the Mainnet
- `index` - [Index API](#) will be enabled
- `ip` - use `dynamic`, `static` arguments, or enter a desired IP directly to be used as the public IP node will advertise to the network
- `rpc` - use `any` or `local` argument to select any or local network interface to be used to listen for RPC calls
- `version` - install a specific node version, instead of the latest. See [here](#) for usage.

Please note that configuring `index` and `archival` options on an existing node will require a fresh bootstrap to recreate the database.

Complete script usage can be displayed by entering:

```
./avalanchego-installer.sh --help
```

## Unattended Installation

If you want to use the script in an automated environment where you cannot enter the data at the prompts you must provide at least the `rpc` and `ip` options. For example:

```
./avalanchego-installer.sh --ip 1.2.3.4 --rpc local
```

## Usage Examples

To run a Fuji node with indexing enabled and autodetected static IP:

```
./avalanchego-installer.sh --fuji --ip static --index
```

To run an archival Mainnet node with dynamic IP and database located at `/home/node/db`:

```
./avalanchego-installer.sh --archival --ip dynamic --db-dir /home/node/db
```

To use C-Chain state-sync to quickly bootstrap a Mainnet node, with dynamic IP and local RPC only:

```
./avalanchego-installer.sh --state-sync on --ip dynamic --rpc local
```

To reinstall the node using node version 1.7.10 and use specific IP and local RPC only:

```
./avalanchego-installer.sh --reinstall --ip 1.2.3.4 --version v1.7.10 --rpc local
```

## Node Configuration

File that configures node operation is `~/.avalanchego/configs/node.json`. You can edit it to add or change configuration options. The documentation of configuration options can be found [here](#). Configuration may look like this:

```
{
 "public-ip-resolution-service": "opendns",
 "http-host": ""
}
```

Note that configuration file needs to be a properly formatted `JSON` file, so switches are formatted differently than for command line, so don't enter options like `--public-ip-resolution-service=opendns` but as in the example above.

Script also creates an empty C-Chain config file, located at `~/.avalanchego/configs/chains/C/config.json`. By editing that file you can configure the C-Chain, as described in detail [here](#).

## Using a Previous Version

The installer script can also be used to install a version of AvalancheGo other than the latest version.

To see a list of available versions for installation, run:

```
./avalanchego-installer.sh --list
```

It will print out a list, something like:

```
AvalancheGo installer

Available versions:
v1.3.2
v1.3.1
v1.3.0
v1.2.4-arm-fix
v1.2.4
v1.2.3-signed
v1.2.3
v1.2.2
v1.2.1
v1.2.0
```

To install a specific version, run the script with `--version` followed by the tag of the version. For example:

```
./avalanchego-installer.sh --version v1.3.1
```

:::danger

Note that not all AvalancheGo versions are compatible. You should generally run the latest version. Running a version other than latest may lead to your node not working properly and, for validators, not receiving a staking reward.

:::

Thanks to community member [Jean Zundel](#) for the inspiration and help implementing support for installing non-latest node versions.

## Reinstall and Script Update

Installer script gets updated from time to time, with new features and capabilities added. To take advantage of new features or to recover from modifications that made the node fail, you may want to reinstall the node. To do that, fetch the latest version of the script from the web with:

```
wget -nd -m https://raw.githubusercontent.com/ava-labs/avalanche-docs/master/scripts/avalanchego-installer.sh
```

After the script has updated, run it again with the `--reinstall` config flag:

```
./avalanchego-installer.sh --reinstall
```

This will delete the existing service file, and run the installer from scratch, like it was started for the first time. Note that the database and NodeID will be left intact.

## Removing the Node Installation

If you want to remove the node installation from the machine, you can run the script with the `--remove` option, like this:

```
./avalanchego-installer.sh --remove
```

This will remove the service, service definition file and node binaries. It will not remove the working directory, node ID definition or the node database. To remove those as well, you can type:

```
rm -rf ~/.avalanche/go/
```

Please note that this is irreversible and the database and node ID will be deleted!

## What Next?

That's it, you're running an AvalancheGo node! Congratulations! Let us know you did it on our [Twitter](#), [Telegram](#) or [Reddit](#)!

If you're on a residential network (dynamic IP), don't forget to set up port forwarding. If you're on a cloud service provider, you're good to go.

Now you can [interact with your node](#), [stake your tokens](#), or level up your installation by setting up [node monitoring](#) to get a better insight into what your node is doing. Also, you might want to use our [Postman Collection](#) to more easily issue commands to your node.

Finally, if you haven't already, it is a good idea to [back up](#) important files in case you ever need to restore your node to a different machine.

**If you have any questions, or need help, feel free to contact us on our [Discord](#) server.**

**sidebar\_position: 3 description: This tutorial will guide you through setting up an Avalanche node on Amazon Web Services (AWS). Cloud services like AWS are a good way to ensure that your node is highly secure, available, and accessible.**

## Run an Avalanche Node with Amazon Web Services (AWS)

### Introduction

This tutorial will guide you through setting up an Avalanche node on [Amazon Web Services \(AWS\)](#). Cloud services like AWS are a good way to ensure that your node is highly secure, available, and accessible.

To get started, you'll need:

- An AWS account
- A terminal with which to SSH into your AWS machine
- A place to securely store and back up files

This tutorial assumes your local machine has a Unix style terminal. If you're on Windows, you'll have to adapt some of the commands used here.

### Log Into AWS

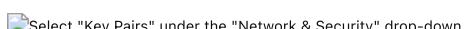
Signing up for AWS is outside the scope of this article, but Amazon has instructions [here](#).

It is *highly* recommended that you set up Multi-Factor Authentication on your AWS root user account to protect it. Amazon has documentation for this [here](#).

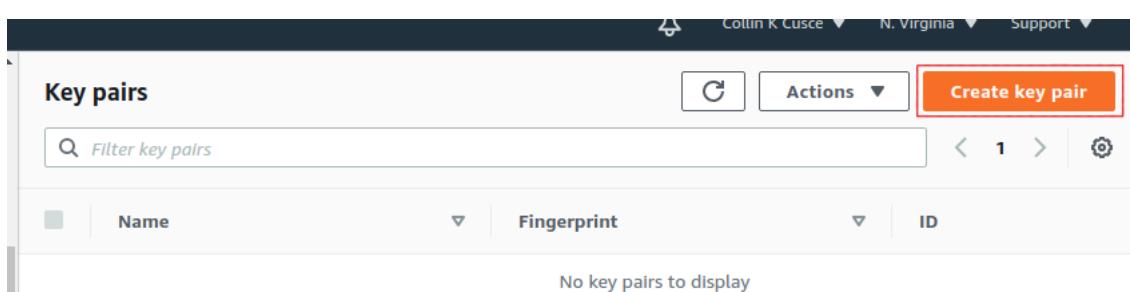
Once your account is set up, you should create a new EC2 instance. An EC2 is a virtual machine instance in AWS's cloud. Go to the [AWS Management Console](#) and enter the EC2 dashboard.



To log into the EC2 instance, you will need a key on your local machine that grants access to the instance. First, create that key so that it can be assigned to the EC2 instance later on. On the bar on the left side, under **Network & Security**, select **Key Pairs**.



Select **Create key pair** to launch the key pair creation wizard.



The screenshot shows the 'Key pairs' section of the AWS Management Console. At the top, there is a search bar labeled 'Filter key pairs' and a navigation bar with icons for 'Actions', 'Create key pair', and other filters. Below the search bar is a table header with columns for 'Name', 'Fingerprint', and 'ID'. The table body displays the message 'No key pairs to display'.

Name your key `avalanche`. If your local machine has MacOS or Linux, select the `pem` file format. If it's Windows, use the `ppk` file format. Optionally, you can add tags for the key pair to assist with tracking.

aws Services ▾ Collin K Cusce ▾

EC2 > Key pairs > Create key pair

## Create key pair

**Key pair**  
A key pair, consisting of a private key and a public key, is a set of security credentials that you use to prove your identity when connecting to an instance.

**Name**  
avalanche  
The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

**File format**  
 pem  
For use with OpenSSH  
 ppk  
For use with PuTTY

**Tags (Optional)**

| Key                       | Value - optional                        |
|---------------------------|-----------------------------------------|
| <input type="text"/> Name | <input type="text"/> Avalanche Key Pair |
| <a href="#">Add tag</a>   |                                         |

You can add 49 more tags

[Cancel](#) [Create key pair](#)

Click [Create key pair](#). You should see a success message, and the key file should be downloaded to your local machine. Without this file, you will not be able to access your EC2 instance. **Make a copy of this file and put it on a separate storage medium such as an external hard drive. Keep this file secret; do not share it with others.**

 Successfully created key pair

| Key pairs (1)                         |           |  |
|---------------------------------------|-----------|-------------------------------------------------------------------------------------|
| <input type="text"/> Filter key pairs |           |                                                                                     |
| <input type="checkbox"/>              | Name      | Fingerprint                                                                         |
| <input type="checkbox"/>              | avalanche | fe:a1:f8:96:19:28:17:a6:ba:                                                         |

### Create a Security Group

An AWS Security Group defines what internet traffic can enter and leave your EC2 instance. Think of it like a firewall. Create a new Security Group by selecting **Security Groups** under the **Network & Security** drop-down.

## ▼ Elastic Block Store

Volumes  
Snapshots  
Lifecycle Manager

## ▼ Network & Security

**Security Groups** [New](#)

Elastic IPs [New](#)  
Placement Groups [New](#)  
Key Pairs [New](#)  
Network Interfaces

This opens the Security Groups panel. Click **Create security group** in the top right of the Security Groups panel.

The screenshot shows the AWS Security Groups list interface. At the top, there's a header with 'Security Groups (1/1)' and a 'Create security group' button highlighted with a red border. Below the header is a search bar labeled 'Filter security groups'. The main table has columns for Name, Security group ID, Security group name, and VPC ID. One row is visible, showing 'sg-38adab41' as the Name, 'sg-38adab41' as the Security group ID, 'default' as the Security group name, and 'vpc-2b52' as the VPC ID. The row has a blue checkmark in the first column.

You'll need to specify what inbound traffic is allowed. Allow SSH traffic from your IP address so that you can log into your EC2 instance (each time your ISP changes your IP address, you will need to modify this rule). Allow TCP traffic on port 9651 so your node can communicate with other nodes on the network. Allow TCP traffic on port 9650 from your IP so you can make API calls to your node. **It's important that you only allow traffic on the SSH and API port from your IP.** If you allow incoming traffic from anywhere, this could be used to brute force entry to your node (SSH port) or used as a denial of service attack vector (API port). Finally, allow all outbound traffic.

 Your inbound and outbound rules should look like this.

Add a tag to the new security group with key `Name` and value `Avalanche Security Group`. This will enable us to know what this security group is when we see it in the list of security groups.

The screenshot shows the 'Tags - optional' configuration page. It includes a descriptive text about tags and a table for adding tags. The table has two columns: 'Key' and 'Value - optional'. A tag for 'Name' with the value 'Avalanche Security Group' is already added. There are buttons for 'Add new tag' and 'Remove'. A note at the bottom says 'You can add up to 49 more tag'.

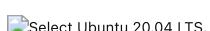
Click **Create security group**. You should see the new security group in the list of security groups.

## Launch an EC2 Instance

Now you're ready to launch an EC2 instance. Go to the EC2 Dashboard and select **Launch instance**.

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with various navigation links like 'Instances', 'Launch Templates', 'Spot Requests', etc. The main area has a blue header bar with the message: 'Welcome to the new EC2 console! We're redesigning the EC2 console to make it easier to use and improve performance. We encourage you to try the new console and the new console, use the New EC2 Experience toggle.' Below this, there's a section titled 'Resources' with stats for Running Instances (0), Snapshots (0), Key pairs (0), Elastic IPs, Volumes, and Security groups. A callout box says: 'Easily size, configure, and deploy Microsoft SQL Server Always On availability groups.' At the bottom, there's a large orange button labeled 'Launch instance ▾'. A note below it says: 'Note: Your instances will launch in the US East (N. Virginia) Region.'

Select Ubuntu 20.04 LTS (HVM), SSD Volume Type for the operating system.



Next, choose your instance type. This defines the hardware specifications of the cloud instance. In this tutorial we set up a c5.2xlarge. This should be more than powerful enough since Avalanche is a lightweight consensus protocol. To create a c5.2xlarge instance, select the **Compute-optimized** option from the filter dropdown menu.

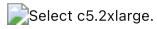
## Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. In this tutorial we will use the **Compute-optimized** instance types to provide the best performance for our application. You can learn more about instance types and how they can meet your specific needs.

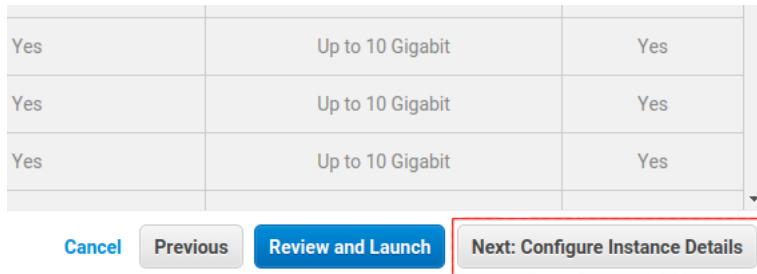
Filter by: All instance types ▾ Current generation ▾ Show/Hide Columns

| Currently                       | Type     | vCPUs |
|---------------------------------|----------|-------|
| All instance types              | t2.nano  | 1     |
| Micro instances                 | t2.micro | 1     |
| General purpose                 | t2.small | 1     |
| Compute optimized               |          |       |
| FPGA instances                  |          |       |
| GPU instances                   |          |       |
| Machine learning ASIC instances |          |       |
| Memory optimized                |          |       |
| Storage optimized               |          |       |

Select the checkbox next to the c5.2xlarge instance in the table.



Click the **Next: Configure Instance Details** button in the bottom right-hand corner.

A screenshot of the AWS EC2 instance configuration page. It shows a table with three rows of network interface options. Below the table are four buttons: "Cancel", "Previous", "Review and Launch", and "Next: Configure Instance Details". The "Next" button is highlighted with a red border.

The instance details can stay as their defaults.

#### Optional: Using Reserved Instances

By default, you will be charged hourly for running your EC2 instance. For a long term usage that is not optimal.

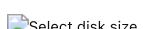
You could save money by using a **Reserved Instance**. With a reserved instance, you pay upfront for an entire year of EC2 usage, and receive a lower per-hour rate in exchange for locking in. If you intend to run a node for a long time and don't want to risk service interruptions, this is a good option to save money. Again, do your own research before selecting this option.

#### Add Storage, Tags, Security Group

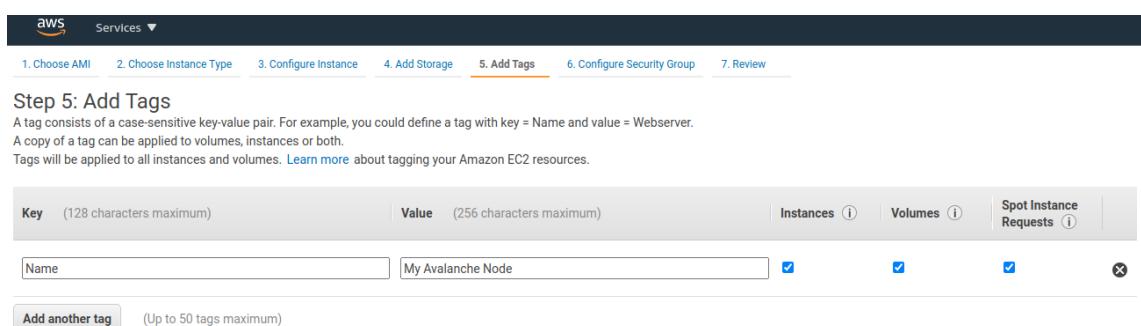
Click the **Next: Add Storage** button in the bottom right corner of the screen.

You need to add space to your instance's disk. You should start with at least 700GB of disk space. Although upgrades to reduce disk usage are always in development, on average the database will continually grow, so you need to constantly monitor disk usage on the node and increase disk space if needed.

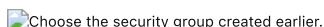
Note that the image below shows 100GB as disk size, which was appropriate at the time the screenshot was taken. You should check the current [recommended disk space size](#) before entering the actual value here.



Click **Next: Add Tags** in the bottom right corner of the screen to add tags to the instance. Tags enable us to associate metadata with our instance. Add a tag with key `Name` and value `My Avalanche Node`. This will make it clear what this instance is on your list of EC2 instances.

A screenshot of the AWS EC2 instance configuration page, specifically the "Step 5: Add Tags" section. It shows a table for adding tags. The first row has "Key" set to "(128 characters maximum)" and "Value" set to "(256 characters maximum)". The second row has "Name" set to "My Avalanche Node" with checkboxes for "Instances", "Volumes", and "Spot Instance Requests" all checked. Below the table is a button "Add another tag" and a note "(Up to 50 tags maximum)".

Now assign the security group created earlier to the instance. Choose **Select an existing security group** and choose the security group created earlier.



Finally, click **Review and Launch** in the bottom right. A review page will show the details of the instance you're about to launch. Review those, and if all looks good, click the blue **Launch** button in the bottom right corner of the screen.

You'll be asked to select a key pair for this instance. Select **Choose an existing key pair** and then select the `avalanche` key pair you made earlier in the tutorial. Check the box acknowledging that you have access to the `.pem` or `.ppk` file created earlier (make sure you've backed it up!) and then click **Launch Instances**.

## Select an existing key pair or create a new key pair

X

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

Choose an existing key pair

Select a key pair

avalanche

I acknowledge that I have access to the selected private key file (avalanche.pem), and that without this file, I won't be able to log into my instance.

Cancel

Launch Instances

You should see a new pop up that confirms the instance is launching!

## Launch Status



Your instances are now launching

The following instance launches have been initiated: [i-04c6e938f127ab994](#) [View launch log](#)

### Assign an Elastic IP

By default, your instance will not have a fixed IP. Let's give it a fixed IP through AWS's Elastic IP service. Go back to the EC2 dashboard. Under **Network & Security**, select **Elastic IPs**.

#### ▼ Network & Security

Security Groups [New](#)

Elastic IPs [New](#)

Placement Groups [New](#)

Key Pairs [New](#)

Network Interfaces

Select **Allocate Elastic IP address**.

The screenshot shows a modal window for allocating an elastic IP. At the top right is a close button (X). Below it are two buttons: 'Actions ▾' and 'Allocate Elastic IP address', with the latter being the one highlighted by a red box. At the bottom are navigation arrows (< 1 >) and a settings gear icon. The main area contains three dropdown menus: 'Associated instance ID' (with a dropdown arrow), 'Private IP address' (with a dropdown arrow), and 'Association ID'.

Select the region your instance is running in, and choose to use Amazon's pool of IPv4 addresses. Click **Allocate**.

# Allocate Elastic IP address

Allocate an Elastic IP address by selecting the public IPv4 address pool from which the public IP address is to be allocated. You can have one Elastic IP (EIP) address associated with a running instance at no charge. If you associate additional EIPs with that instance, you will be charged for each additional EIP associated with that instance on a pro rata basis. Additional EIPs are only available in Amazon VPC. To ensure efficient use of Elastic IP addresses, we impose a small hourly charge when these IP addresses are not associated with a running instance or when they are associated with a stopped instance or unattached network interface. [Learn more](#)

## Elastic IP address settings

**Network Border Group**  
A Network Border Group is a logical group of Zones from where public IPv4 addresses are advertised. Set this parameter to limit the IPv4 address to the Zones in Network Border Group.

X

**Public IPv4 address pool**  
Public IP addresses are allocated from Amazon's pool of public IP addresses, from a pool that you own and bring to your account, or from a pool that you own and continue to advertise..

- Amazon's pool of IPv4 addresses
- Public IPv4 address that you bring to your AWS account(option disabled because no pools found) [Learn more](#)
- Customer owned pool of IPv4 addresses(option disabled because no customer owned pools found) [Learn more](#)

Cancel Allocate

Select the Elastic IP you just created from the Elastic IP manager. From the **Actions** drop-down, choose **Associate Elastic IP address**.

The screenshot shows a table of Elastic IP resources. The first column is labeled "Associated instance ID". The second column contains three actions: "View details", "Release Elastic IP addresses", and "Associate Elastic IP address". The "Associate Elastic IP address" option is highlighted with a red box. The third column is labeled "Association ID" and contains a single entry: "-". The "Actions" button is also highlighted with a red box.

Select the instance you just created. This will associate the new Elastic IP with the instance and give it a public IP address that won't change.

## Associate Elastic IP address

Choose the instance or network interface to associate to this Elastic IP address (54.83.240.159)

**Elastic IP address: 54.83.240.159**

**Resource type**  
Choose the type of resource with which to associate the Elastic IP address.

Instance

Network interface

**⚠️** If you associate an Elastic IP address to an instance that already has an Elastic IP address associated, this previously associated Elastic IP address will be disassociated but still allocated to your account. [Learn more](#)

**Instance**  
 X C

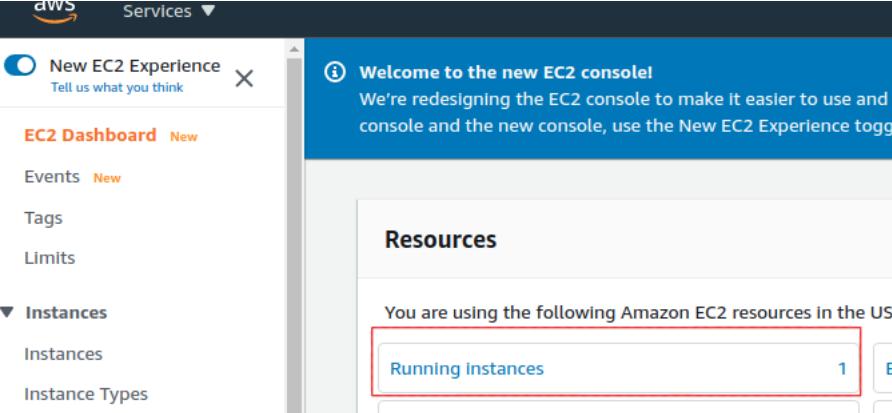
**Private IP address**  
The private IP address with which to associate the Elastic IP address.  
 X

**Reassociation**  
Specify whether the Elastic IP address can be reassigned to a different resource if it's already associated with a resource.  
 Allow this Elastic IP address to be reassigned

Cancel Associate

## Set Up AvalancheGo

Go back to the EC2 Dashboard and select `Running Instances`.



The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with options like 'New EC2 Experience', 'EC2 Dashboard', 'Events', 'Tags', 'Limits', and 'Instances'. Under 'Instances', there are links for 'Instances' and 'Instance Types'. The main area has a blue header bar with the text 'Welcome to the new EC2 console!'. Below this, there's a section titled 'Resources' with the sub-section 'Running Instances'. A red box highlights the 'Running Instances' link. To the right of the resources section, there's a small icon with the number '1' and a letter 'E'.

Select the newly created EC2 instance. This opens a details panel with information about the instance.

 Details about your new instance.

Copy the `IPv4 Public IP` field to use later. From now on we call this value `PUBLICIP`.

**Remember:** the terminal commands below assume you're running Linux. Commands may differ for MacOS or other operating systems. When copy-pasting a command from a code block, copy and paste the entirety of the text in the block.

Log into the AWS instance from your local machine. Open a terminal (try shortcut **CTRL + ALT + T**) and navigate to the directory containing the `.pem` file you downloaded earlier.

Move the `.pem` file to `$HOME/.ssh` (where `.pem` files generally live) with:

```
mv avalanche.pem ~/.ssh
```

Add it to the SSH agent so that we can use it to SSH into your EC2 instance, and mark it as read-only.

```
ssh-add ~/.ssh/avalanche.pem; chmod 400 ~/.ssh/avalanche.pem
```

SSH into the instance. (Remember to replace `PUBLICIP` with the public IP field from earlier.)

```
ssh ubuntu@PUBLICIP
```

If the permissions are **not** set correctly, you will see the following error.

```
ccusce@AVAccuse:~/tutorials/aws Validators$ ssh -i avalanche.pem ubuntu@ec2-3-88-204-20.compute-1.amazonaws.com
The authenticity of host 'ec2-3-88-204-20.compute-1.amazonaws.com (3.88.204.20)' can't be established.
ECDSA key fingerprint is SHA256:IDIO5Fm+4oNXFgxf4gdTdqf9Y3XWd0oMgrb8pKNxjI.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-3-88-204-20.compute-1.amazonaws.com,3.88.204.20' (ECDSA) to the list of known hosts.
@@@@@@@WARNING: UNPROTECTED PRIVATE KEY FILE! @@@
@@@Permissions 0644 for 'avalanche.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "avalanche.pem": bad permissions
ubuntu@ec2-3-88-204-20.compute-1.amazonaws.com: Permission denied (publickey).
ccusce@AVAccuse:~/tutorials/aws Validators$
```

You are now logged into the EC2 instance.

```
ccusce@AVAccuse:~/tutorials/aws Validators$ chmod 400 avalanche.pem
ccusce@AVAccuse:~/tutorials/aws Validators$ ssh -i avalanche.pem ubuntu@ec2-3-88-204-20.compute-1.amazonaws.com
Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.3.0-1035-aws x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

 System information as of Tue Sep 22 20:18:30 UTC 2020

 System load: 0.0 Processes: 93
 Usage of /: 2.9% of 38.71GB Users logged in: 0
 Memory usage: 5%
 Swap usage: 0%
 IP address for ens5: 172.31.10.137

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-10-137:~$
```

If you have not already done so, update the instance to make sure it has the latest operating system and security updates:

```
sudo apt update; sudo apt upgrade -y; sudo reboot
```

This also reboots the instance. Wait 5 minutes, then log in again by running this command on your local machine:

```
ssh ubuntu@PUBLICIP
```

You're logged into the EC2 instance again. Now we'll need to set up our Avalanche node. To do this, follow the [Set Up Avalanche Node With Installer](#) tutorial which automates the installation process. You will need the `PUBLICIP` we set up earlier.

Your AvalancheGo node should now be running and in the process of bootstrapping, which can take a few hours. To check if it's done, you can issue an API call using `curl`. If you're making the request from the EC2 instance, the request is:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.isBootstrapped",
 "params": {
 "chain": "X"
 }
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

Once the node is finished bootstrapping, the response will be:

```
{
 "jsonrpc": "2.0",
 "result": {
 "isBootstrapped": true
 },
 "id": 1
}
```

You can continue on, even if AvalancheGo isn't done bootstrapping.

In order to make your node a validator, you'll need its node ID. To get it, run:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

The response contains the node ID.

```
{"jsonrpc": "2.0", "result": {"nodeID": "NodeID-DznHm3o7RkmpLkWMn9NqafH66mqunXbM"}, "id": 1}
```

In the above example the node ID is `NodeID-DznHm3o7RkmpLkWMn9NqafH66mqunXbM`. Copy your node ID for later. Your node ID is not a secret, so you can just paste it into a text editor.

AvalancheGo has other APIs, such as the [Health API](#), that may be used to interact with the node. Some APIs are disabled by default. To enable such APIs, modify the ExecStart section of `/etc/systemd/system/avalanchego.service` (created during the installation process) to include flags that enable these endpoints. Don't manually enable any APIs unless you have a reason to.

```
Usage of avalanchego:
-api-admin-enabled
 If true, this node exposes the Admin API
-api-auth-password string
 Password used to create/validate API authorization tokens. Can be changed via API call.
-api-auth-required
 Require authorization token to call HTTP APIs
-api-health-enabled
 If true, this node exposes the Health API (default true)
-api-info-enabled
 If true, this node exposes the Info API (default true)
-api-ipcs-enabled
 If true, IPCs can be opened
-api-keystore-enabled
 If true, this node exposes the Keystore API (default true)
-api-metrics-enabled
 If true, this node exposes the Metrics API (default true)
```

Back up the node's staking key and certificate in case the EC2 instance is corrupted or otherwise unavailable. The node's ID is derived from its staking key and certificate. If you lose your staking key or certificate then your node will get a new node ID, which could cause you to become ineligible for a staking reward if your node is a validator. **It is very strongly advised that you copy your node's staking key and certificate.** The first time you run a node, it will generate a new staking key/certificate pair and store them in directory `/home/ubuntu/.avalanchego/staking`.

Exit out of the SSH instance by running:

```
exit
```

Now you're no longer connected to the EC2 instance; you're back on your local machine.

To copy the staking key and certificate to your machine, run the following command. As always, replace `PUBLICIP`.

```
scp -r ubuntu@PUBLICIP:/home/ubuntu/.avalanchego/staking ~/aws_avalanche_backup
```

Now your staking key and certificate are in directory `~/aws_avalanche_backup`. **The contents of this directory are secret.** You should hold this directory on storage not connected to the internet (like an external hard drive.)

### Upgrading Your Node

AvalancheGo is an ongoing project and there are regular version upgrades. Most upgrades are recommended but not required. Advance notice will be given for upgrades that are not backwards compatible. To update your node to the latest version, SSH into your AWS instance as before and run the installer script again.

```
./avalanchego-installer.sh
```

Your machine is now running the newest AvalancheGo version. To see the status of the AvalancheGo service, run `sudo systemctl status avalanchego`.

### Increase Volume Size

If you need to increase the volume size, follow these instructions from AWS:

- [Request modifications to your EBS volumes](#)
- [Extend a Linux file system after resizing a volume](#)

### Wrap Up

**That's it! You now have an AvalancheGo node running on an AWS EC2 instance. We recommend setting up [node monitoring](#) for your AvalancheGo node. We also recommend setting up AWS billing alerts so you're not surprised when the bill arrives. If you have feedback on this tutorial, or anything else, send us a message on [Discord](#).**

**sidebar\_position: 9**

## Avalanche Notify

To receive email alerts if a validator becomes unresponsive or out-of-date, sign up with the Avalanche Notify tool: <http://notify.avax.network>.

Avalanche Notify is an active monitoring system that checks a validator's responsiveness each minute.

An email alert is sent if a validator is down for 5 consecutive checks and when a validator recovers (is responsive for 5 checks in a row).

:::tip

When signing up for email alerts, consider using a new, alias, or auto-forwarding email address to protect your privacy. Otherwise, it will be possible to link your NodeID to your email.

:::

:::warning Disclaimer

This tool is currently in BETA and validator alerts may erroneously be triggered, not triggered, or delayed. The best way to maximize the likelihood of earning staking rewards is to run redundant monitoring/alerting.

:::

**sidebar\_position: 4 description: This documents list all available configuration and flags for AvalancheGo.**

## AvalancheGo Config and Flags

You can specify the configuration of a node with the arguments below.

### Data Directory

`--data-dir (string)`

Sets the base data directory where default sub-directories will be placed unless otherwise specified. Defaults to `$HOME/.avalanchego`.

### Config File

`--config-file (string)`

Path to a JSON file that specifies this node's configuration. Command line arguments will override arguments set in the config file. This flag is ignored if `--config-file-content` is specified.

Example JSON config file:

```
{
 "log-level": "debug"
}
```

:::tip [Install Script](#) creates the node config file at `~/.avalanchego/configs/node.json`. No default file is created if [AvalancheGo is built from source](#), you would need to create it manually if needed. :::

#### `--config-file-content (string)`

As an alternative to `--config-file`, it allows specifying base64 encoded config content.

#### `--config-file-content-type (string)`

Specifies the format of the base64 encoded config content. JSON, TOML, YAML are among currently supported file format (see [here](#) for full list). Defaults to `JSON`.

## APIs

#### `--api-admin-enabled (boolean)`

If set to `true`, this node will expose the Admin API. Defaults to `false`. See [here](#) for more information.

#### `--api-auth-required (boolean)`

If set to `true`, API calls require an authorization token. Defaults to `false`. See [here](#) for more information.

#### `--api-auth-password (string)`

The password needed to create/revoke authorization tokens. If `--api-auth-required=true`, must be specified; otherwise ignored. See [here](#) for more information.

#### `--api-auth-password-file (string)`

Password file used to initially create/validate API authorization tokens. Ignored if `--api-auth-password` is specified. Leading and trailing whitespace is removed from the password. Can be changed via API call.

#### `--api-health-enabled (boolean)`

If set to `false`, this node will not expose the Health API. Defaults to `true`. See [here](#) for more information.

#### `--index-enabled (boolean)`

If set to `true`, this node will enable the indexer and the Index API will be available. Defaults to `false`. See [here](#) for more information.

#### `--api-info-enabled (boolean)`

If set to `false`, this node will not expose the Info API. Defaults to `true`. See [here](#) for more information.

#### `--api-ipcs-enabled (boolean)`

If set to `true`, this node will expose the IPCs API. Defaults to `false`. See [here](#) for more information.

#### `--api-keystore-enabled (boolean)`

If set to `true`, this node will expose the Keystore API. Defaults to `false`. See [here](#) for more information.

#### `--api-metrics-enabled (boolean)`

If set to `false`, this node will not expose the Metrics API. Defaults to `true`. See [here](#) for more information.

#### `--http-shutdown-wait (duration)`

Duration to wait after receiving SIGTERM or SIGINT before initiating shutdown. The `/health` endpoint will return unhealthy during this duration (if the Health API is enabled.) Defaults to `0s`.

#### `--http-shutdown-timeout (duration)`

Maximum duration to wait for existing connections to complete during node shutdown. Defaults to `10s`.

## Bootstrapping

#### `--bootstrap-beacon-connection-timeout (duration)`

Timeout when attempting to connect to bootstrapping beacons. Defaults to `1m`.

#### `--bootstrap-ids (string)`

Bootstrap IDs is a comma-separated list of validator IDs. These IDs will be used to authenticate bootstrapping peers. An example setting of this field would be `--bootstrap-ids="NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg,NodeID-MFrZFVcXPv5iCn6M9K6XduxGTYp891xxZ"`. The number of given IDs here must be same with number of given `--bootstrap-ips`. The default value depends on the network ID.

#### `--bootstrap-ips (string)`

Bootstrap IPs is a comma-separated list of IP:port pairs. These IP Addresses will be used to bootstrap the current Avalanche state. An example setting of this field would be `--bootstrap-ips="127.0.0.1:12345,1.2.3.4:5678"`. The number of given IPs here must be same with number of given `--bootstrap-ids`. The default value depends on the network ID.

#### `--bootstrap-retry-enabled (boolean)`

If set to `false`, will not retry bootstrapping if it fails. Defaults to `true`.

#### `--bootstrap-retry-warn-frequency (uint)`

Specifies how many times bootstrap should be retried before warning the operator. Defaults to `50`.

#### `--bootstrap-ancestors-max-containers-sent (uint)`

Max number of containers in an `Ancestors` message sent by this node. Defaults to `2000`.

#### `--bootstrap-ancestors-max-containers-received (unit)`

This node reads at most this many containers from an incoming `Ancestors` message. Defaults to `2000`.

#### `--bootstrap-max-time-get-ancestors (duration)`

Max Time to spend fetching a container and its ancestors when responding to a GetAncestors message. Defaults to `50ms`.

## State Syncing

#### `--state-sync-ids (string)`

State sync IDs is a comma-separated list of validator IDs. The specified validators will be contacted to get and authenticate the starting point (state summary) for state sync. An example setting of this field would be `--state-sync-ids="NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg,NodeID-MFrZFVCXPv5iCn6M9K6XduxGTYp891xXZ"`. The number of given IDs here must be same with number of given `--state-sync-ips`. The default value is empty, which results in all validators being sampled.

#### `--state-sync-ips (string)`

State sync IPs is a comma-separated list of IP:port pairs. These IP Addresses will be contacted to get and authenticate the starting point (state summary) for state sync. An example setting of this field would be `--state-sync-ips="127.0.0.1:12345,1.2.3.4:5678"`. The number of given IPs here must be the same with the number of given `--state-sync-ids`.

## Chain Configs

Some blockchains allow the node operator to provide custom configurations for individual blockchains. These custom configurations are broken down into two categories: network upgrades and optional chain configurations. AvalancheGo reads in these configurations from the chain configuration directory and passes them into the VM on initialization.

#### `--chain-config-dir (string)`

Specifies the directory that contains chain configs, as described [here](#). Defaults to `$HOME/.avalanchego/configs/chains`. If this flag is not provided and the default directory does not exist, AvalancheGo will not exit since custom configs are optional. However, if the flag is set, the specified folder must exist, or AvalancheGo will exit with an error. This flag is ignored if `--chain-config-content` is specified.

:::note Please replace `chain-config-dir` and `blockchainID` with their actual values. :::

Network upgrades are passed in from the location: `chain-config-dir / blockchainID / upgrade.*`. Upgrade files are typically json encoded and therefore named `upgrade.json`. However, the format of the file is VM dependent. After a blockchain has activated a network upgrade, the same upgrade configuration must always be passed in to ensure that the network upgrades activate at the correct time.

The chain configs are passed in from the location `chain-config-dir / blockchainID / config.*`. Upgrade files are typically json encoded and therefore named `upgrade.json`. However, the format of the file is VM dependent. This configuration is used by the VM to handle optional configuration flags such as enabling/disabling APIs, updating log level, etc. The chain configuration is intended to provide optional configuration parameters and the VM will use default values if nothing is passed in.

Full reference for all configuration options for some standard chains can be found in a separate [chain config flags](#) document.

Full reference for `subnet-evm` upgrade configuration can be found in a separate [Customize a Subnet](#) document.

#### `--chain-config-content (string)`

As an alternative to `--chain-config-dir`, chains custom configurations can be loaded altogether from command line via `--chain-config-content` flag. Content must be base64 encoded.

#### `--chain-aliases-file (string)`

Path to JSON file that defines aliases for Blockchain IDs. Defaults to `~/.avalanchego/configs/chains/aliases.json`. This flag is ignored if `--chain-aliases-file-content` is specified. Example content:

```
{
 "q2aTwKuyzgs8pynF7UXBZCU7DejbzbZ6EUyHr3JQzYgwNPUPi": ["DFK"]
}
```

The above example aliases the Blockchain whose ID is `"q2aTwKuyzgs8pynF7UXBZCU7DejbzbZ6EUyHr3JQzYgwNPUPi"` to `"DFK"`. Chain aliases are added after adding primary network aliases and before any changes to the aliases via the admin API. This means that the first alias included for a Blockchain on a Subnet will be treated as the `"Primary Alias"` instead of the full blockchainID. The Primary Alias is used in all metrics and logs.

#### `--chain-aliases-file-content (string)`

As an alternative to `--chain-aliases-file`, it allows specifying base64 encoded aliases for Blockchains.

```
--chain-data-dir (string)
Chain specific data directory. Defaults to $HOME/.avalanchego/chainData .
```

## Database

```
--db-dir (string, file path)
```

Specifies the directory to which the database is persisted. Defaults to "\$HOME/.avalanchego/db" .

```
--db-type (string)
```

Specifies the type of database to use. Must be one of LevelDB or memdb . memdb is an in-memory, non-persisted database.

:::note

memdb stores everything in memory. So if you have a 900 GiB LevelDB instance, then using memdb you'd need 900 GiB of RAM. memdb is useful for fast one-off testing, not for running an actual node (on Fuji or Mainnet). Also note that memdb doesn't persist after restart. So any time you restart the node it would start syncing from scratch.

:::

## Database Config

```
--db-config-file (string)
```

Path to the database config file. Ignored if --config-file-content is specified.

```
--db-config-file-content (string)
```

As an alternative to --db-config-file , it allows specifying base64 encoded database config content.

## LevelDB Config

A LevelDB config file must be JSON and may have these keys. Any keys not given will receive the default value.

```
{
 // BlockCacheCapacity defines the capacity of the 'sorted table' block caching.
 // Use -1 for zero.
 //
 // The default value is 12MiB.
 "blockCacheCapacity": int

 // BlockSize is the minimum uncompressed size in bytes of each 'sorted table'
 // block.
 //
 // The default value is 4KiB.
 "blockSize": int

 // CompactionExpandLimitFactor limits compaction size after expanded.
 // This will be multiplied by table size limit at compaction target level.
 //
 // The default value is 25.
 "compactionExpandLimitFactor": int

 // CompactionGPOverlapsFactor limits overlaps in grandparent (Level + 2)
 // that a single 'sorted table' generates. This will be multiplied by
 // table size limit at grandparent level.
 //
 // The default value is 10.
 "compactionGPOverlapsFactor": int

 // CompactionL0Trigger defines number of 'sorted table' at level-0 that will
 // trigger compaction.
 //
 // The default value is 4.
 "compactionL0Trigger": int

 // CompactionSourceLimitFactor limits compaction source size. This doesn't apply to
 // level-0.
 // This will be multiplied by table size limit at compaction target level.
 //
 // The default value is 1.
 "compactionSourceLimitFactor": int

 // CompactionTableSize limits size of 'sorted table' that compaction generates.
 // The limits for each level will be calculated as:
 // CompactionTableSize * (CompactionTableSizeMultiplier ^ Level)
 // The multiplier for each level can also fine-tuned using CompactionTableSizeMultiplierPerLevel.
 //
 // The default value is 2MiB.
 "compactionTableSize": int
```

```

// CompactionTableSizeMultiplier defines multiplier for CompactionTableSize.
//
// The default value is 1.
"compactionTableSizeMultiplier": float

// CompactionTableSizeMultiplierPerLevel defines per-level multiplier for
// CompactionTableSize.
// Use zero to skip a level.
//
// The default value is nil.
"compactionTableSizeMultiplierPerLevel": []float

// CompactionTotalSize limits total size of 'sorted table' for each level.
// The limits for each level will be calculated as:
// CompactionTotalSize * (CompactionTotalSizeMultiplier ^ Level)
// The multiplier for each level can also fine-tuned using
// CompactionTotalSizeMultiplierPerLevel.
//
// The default value is 10MiB.
"compactionTotalSize": int

// CompactionTotalSizeMultiplier defines multiplier for CompactionTotalSize.
//
// The default value is 10.
"compactionTotalSizeMultiplier": float

// DisableSeeksCompaction allows disabling 'seeks triggered compaction'.
// The purpose of 'seeks triggered compaction' is to optimize database so
// that 'level seeks' can be minimized, however this might generate many
// small compaction which may not preferable.
//
// The default is true.
"disableSeeksCompaction": bool

// OpenFilesCacheCapacity defines the capacity of the open files caching.
// Use -1 for zero, this has same effect as specifying NoCacher to OpenFilesCacher.
//
// The default value is 1024.
"openFilesCacheCapacity": int

// WriteBuffer defines maximum size of a 'memdb' before flushed to
// 'sorted table'. 'memdb' is an in-memory DB backed by an on-disk
// unsorted journal.
//
// LevelDB may hold up to two 'memdb' at the same time.
//
// The default value is 6MiB.
"writeBuffer": int

// FilterBitsPerKey is the number of bits to add to the bloom filter per
// key.
//
// The default value is 10.
"filterBitsPerKey": int

// MaxManifestFileSize is the maximum size limit of the MANIFEST***** file.
// When the MANIFEST***** file grows beyond this size, LevelDB will create
// a new MANIFEST file.
//
// The default value is infinity.
"maxManifestFileSize": int

// MetricUpdateFrequency is the frequency to poll LevelDB metrics in
// nanoseconds.
// If <= 0, LevelDB metrics aren't polled.
//
// The default value is 10s.
"metricUpdateFrequency": int
}

```

## Genesis

--genesis-file (string)

Path to a JSON file containing the genesis data to use. Ignored when running standard networks (Mainnet, Fuji Testnet), or when --genesis-content is specified. If not given, uses default genesis data.

These are the main properties in the JSON file:

- `networkID` : A unique identifier for the blockchain, must be a number in the range [0, 2^32].
- `allocations` : The list of initial addresses, their initial balances and the unlock schedule for each.
- `startTime` : The time of the beginning of the blockchain, it must be a Unix timestamp and it can't be a time in the future.
- `initialStakeDuration` : The stake duration, in seconds, of the validators that exist at network genesis.
- `initialStakeDurationOffset` : The offset, in seconds, between the start times of the validators that exist at genesis.
- `initialStakedFunds` : A list of addresses that own the funds staked at genesis (each address must be present in `allocations` as well)
- `initialStakers` : The validators that exist at genesis. Each element contains the `rewardAddress`, `NodeID` and the `delegationFee` of the validator.
- `cChainGenesis` : The genesis info to be passed to the C-Chain.
- `message` : A message to include in the genesis. Not required.

For an example of a JSON representation of genesis data, see [genesis\\_local.json](#).

#### `--genesis (string)`

This flag is deprecated as of v1.10.2. Use `--genesis-file` instead.

#### `--genesis-file-content (string)`

As an alternative to `--genesis-file`, it allows specifying base64 encoded genesis data to use.

#### `--genesis-content (string)`

This flag is deprecated as of v1.10.2. Use `--genesis-file-content` instead.

## HTTP Server

#### `--http-host (string)`

The address that HTTP APIs listen on. Defaults to `127.0.0.1`. This means that by default, your node can only handle API calls made from the same machine. To allow API calls from other machines, use `--http-host=`. You can also enter domain names as parameter.

#### `--http-port (int)`

Each node runs an HTTP server that provides the APIs for interacting with the node and the Avalanche network. This argument specifies the port that the HTTP server will listen on. The default value is `9650`.

#### `--http-tls-cert-file (string, file path)`

This argument specifies the location of the TLS certificate used by the node for the HTTPS server. This must be specified when `--http-tls-enabled=true`. There is no default value. This flag is ignored if `--http-tls-cert-file-content` is specified.

#### `--http-tls-cert-file-content (string)`

As an alternative to `--http-tls-cert-file`, it allows specifying base64 encoded content of the TLS certificate used by the node for the HTTPS server. Note that full certificate content, with the leading and trailing header, must be base64 encoded. This must be specified when `--http-tls-enabled=true`.

#### `--http-tls-enabled (boolean)`

If set to `true`, this flag will attempt to upgrade the server to use HTTPS. Defaults to `false`.

#### `--http-tls-key-file (string, file path)`

This argument specifies the location of the TLS private key used by the node for the HTTPS server. This must be specified when `--http-tls-enabled=true`. There is no default value. This flag is ignored if `--http-tls-key-file-content` is specified.

#### `--http-tls-key-file-content (string)`

As an alternative to `--http-tls-key-file`, it allows specifying base64 encoded content of the TLS private key used by the node for the HTTPS server. Note that full private key content, with the leading and trailing header, must be base64 encoded. This must be specified when `--http-tls-enabled=true`.

#### `--http-read-timeout (string)`

Maximum duration for reading the entire request, including the body. A zero or negative value means there will be no timeout.

#### `--http-read-header-timeout (string)`

Maximum duration to read request headers. The connection's read deadline is reset after reading the headers. If `--http-read-header-timeout` is zero, the value of `--http-read-timeout` is used. If both are zero, there is no timeout.

#### `--http-write-timeout (string)`

Maximum duration before timing out writes of the response. It is reset whenever a new request's header is read. A zero or negative value means there will be no timeout.

#### `--http-idle-timeout (string)`

Maximum duration to wait for the next request when keep-alives are enabled. If `--http-idle-timeout` is zero, the value of `--http-read-timeout` is used. If both are zero, there is no timeout.

#### `--http-allowed-origins (string)`

Origins to allow on the HTTP port. Defaults to `*` which allows all origins. Example: `"https://*.avax.network https://*.avax-test.network"`

#### --http-allowed-hosts (string)

List of acceptable host names in API requests. Provide the wildcard ( '\*' ) to accept requests from all hosts. API requests where the `Host` field is empty or an IP address will always be accepted. An API call whose HTTP `Host` field isn't acceptable will receive a 403 error code. Defaults to `localhost`.

## IPCs

#### --ipcs-chain-ids (string)

Comma separated list of chain ids to connect to (for example

`11111111111111111111111111111111LpoYY,4R5p2RXDGLqaifZE4hHWH9owe34pfoBULn1DrQTWivjg8o4aH`). There is no default value.

#### --ipcs-path (string)

The directory (Unix) or named pipe prefix (Windows) for IPC sockets. Defaults to `/tmp`.

## File Descriptor Limit

#### --fd-limit (int)

Attempts to raise the process file descriptor limit to at least this value and error if the value is above the system max. Linux default `32768`.

## Logging

#### --log-level (string, {verb, debug, trace, info, warn, error, fatal, off})

The log level determines which events to log. There are 8 different levels, in order from highest priority to lowest.

- `off` : No logs have this level of logging. Turns off logging.
- `fatal` : Fatal errors that are not recoverable.
- `error` : Errors that the node encounters, these errors were able to be recovered.
- `warn` : A Warning that might be indicative of a spurious byzantine node, or potential future error.
- `info` : Useful descriptions of node status updates.
- `trace` : Traces container (block, vertex, transaction) job results. Useful for tracing container IDs and their outcomes.
- `debug` : Debug logging is useful when attempting to understand possible bugs in the code. More information that would be typically desired for normal usage will be displayed.
- `verb` : Tracks extensive amounts of information the node is processing. This includes message contents and binary dumps of data for extremely low level protocol analysis.

When specifying a log level note that all logs with the specified priority or higher will be tracked. Defaults to `info`.

#### --log-display-level (string, {verb, debug, trace, info, warn, error, fatal, off})

The log level determines which events to display to stdout. If left blank, will default to the value provided to `--log-level`.

#### --log-format (string, {auto, plain, colors, json})

The structure of log format. Defaults to `auto` which formats terminal-like logs, when the output is a terminal. Otherwise, should be one of `{auto, plain, colors, json}`

#### --log-dir (string, file path)

Specifies the directory in which system logs are kept. Defaults to `"$HOME/.avalanchego/logs"`. If you are running the node as a system service (ex. using the installer script) logs will also be stored in `$HOME/var/log/syslog`.

#### --log-disable-display-plugin-logs (boolean)

Disables displaying plugin logs in stdout. Defaults to `false`.

#### --log-rotater-max-size (uint)

The maximum file size in megabytes of the log file before it gets rotated. Defaults to `8`.

#### --log-rotater-max-files (uint)

The maximum number of old log files to retain. 0 means retain all old log files. Defaults to `7`.

#### --log-rotater-max-age (uint)

The maximum number of days to retain old log files based on the timestamp encoded in their filename. 0 means retain all old log files. Defaults to `0`.

#### --log-rotater-compress-enabled (boolean)

Enables the compression of rotated log files through gzip. Defaults to `false`.

## Network ID

#### --network-id (string)

The identity of the network the node should connect to. Can be one of:

- `--network-id=mainnet` -> Connect to Mainnet (default).
- `--network-id=fuji` -> Connect to the Fuji test-network.

- `--network-id=testnet` -> Connect to the current test-network. (Right now, this is Fuji.)
- `--network-id=local` -> Connect to a local test-network.
- `--network-id=network-(id)` -> Connect to the network with the given ID. `id` must be in the range `[0, 2^32]`.

## OpenTelemetry

AvalancheGo supports collecting and exporting [OpenTelemetry](#) traces. This might be useful for debugging, performance analysis, or monitoring.

### `--tracing-enabled (boolean)`

If true, enable OpenTelemetry tracing. Defaults to `false`.

### `--tracing-endpoint (string)`

The endpoint to export trace data to. Defaults to `localhost:4317`.

### `--tracing-insecure (string)`

If true, don't use TLS when exporting trace data. Defaults to `true`.

### `--tracing-sample-rate (float)`

The fraction of traces to sample. If  $\geq 1$ , always sample. If  $\leq 0$ , never sample. Defaults to `0.1`.

### `--tracing-exporter-type (string)`

Type of exporter to use for tracing. Options are `[ grpc , http ]`. Defaults to `grpc`.

## Public IP

Validators must know one of their public facing IP addresses so they can enable other nodes to connect to them.

By default, the node will attempt to perform NAT traversal to get the node's IP according to its router.

### `--public-ip (string)`

If this argument is provided, the node assume this is its public IP.

:::tip When running a local network it may be easiest to set this value to `127.0.0.1` :::

### `--public-ip-resolution-frequency (duration)`

Frequency at which this node resolves/updates its public IP and renew NAT mappings, if applicable. Default to 5 minutes.

### `--public-ip-resolution-service (string)`

When provided, the node will use that service to periodically resolve/update its public IP. Only acceptable values are `ifconfigCo`, `opendns` or `ifconfigMe`.

## Staking

### `--staking-port (int)`

The port through which the network peers will connect to this node externally. Having this port accessible from the internet is required for correct node operation. Defaults to `9651`.

### `--sybil-protection-enabled (boolean)`

Avalanche uses Proof of Stake (PoS) as sybil resistance to make it prohibitively expensive to attack the network. If false, sybil resistance is disabled and all peers will be sampled during consensus. Defaults to `true`. Note that this can not be disabled on public networks (`Fuji` and `Mainnet`).

Setting this flag to `false` **does not** mean "this node is not a validator." It means that this node will sample all nodes, not just validators. **You should not set this flag to false unless you understand what you are doing.**

### `--staking-enabled (boolean)`

This flag is deprecated as of v1.10.2. Use `--sybil-protection-enabled` instead.

### `--sybil-protection-disabled-weight (uint)`

Weight to provide to each peer when staking is disabled. Defaults to `100`.

### `--staking-disabled-weight (uint)`

This flag is deprecated as of v1.10.2. Use `--sybil-protection-disabled-weight` instead.

### `--staking-tls-cert-file (string, file path)`

Avalanche uses two-way authenticated TLS connections to securely connect nodes. This argument specifies the location of the TLS certificate used by the node. By default, the node expects the TLS certificate to be at `$HOME/.avalanchego/staking/staker.crt`. This flag is ignored if `--staking-tls-cert-file-content` is specified.

### `--staking-tls-cert-file-content (string)`

As an alternative to `--staking-tls-cert-file`, it allows specifying base64 encoded content of the TLS certificate used by the node. Note that full certificate content, with the leading and trailing header, must be base64 encoded.

#### --staking-tls-key-file (string, file path)

Avalanche uses two-way authenticated TLS connections to securely connect nodes. This argument specifies the location of the TLS private key used by the node. By default, the node expects the TLS private key to be at `$HOME/.avalanchego/staking/staker.key`. This flag is ignored if `--staking-tls-key-file-content` is specified.

#### --staking-tls-key-file-content (string)

As an alternative to `--staking-tls-key-file`, it allows specifying base64 encoded content of the TLS private key used by the node. Note that full private key content, with the leading and trailing header, must be base64 encoded.

## Subnets

### Subnet Tracking

#### --track-subnets (string)

Comma separated list of Subnet IDs that this node would track if added to. Defaults to empty (will only validate the Primary Network).

### Subnet Configs

It is possible to provide parameters for Subnets. Parameters here apply to all chains in the specified Subnets. Parameters must be specified with a `{subnetID}.json` config file under `--subnet-config-dir`. AvalancheGo loads configs for Subnets specified in `--track-subnets` parameter.

Full reference for all configuration options for a Subnet can be found in a separate [Subnet Configs](#) document.

#### --subnet-config-dir (string)

Specifies the directory that contains Subnet configs, as described above. Defaults to `$HOME/.avalanchego/configs/subnets`. If the flag is set explicitly, the specified folder must exist, or AvalancheGo will exit with an error. This flag is ignored if `--subnet-config-content` is specified.

Example: Let's say we have a Subnet with ID `p4jUwqZsA2LuSftr0Cd3zb4ytH8W99oXKuKVZdsty7eQ3rXD6`. We can create a config file under the default `subnet-config-dir` at `$HOME/.avalanchego/configs/subnets/p4jUwqZsA2LuSftr0Cd3zb4ytH8W99oXKuKVZdsty7eQ3rXD6.json`. An example config file is:

```
{
 "validatorOnly": false,
 "consensusParameters": {
 "k": 25,
 "alpha": 18
 },
 "appGossipNonValidatorSize": 10
}
```

:::tip By default, none of these directories and/or files exist. You would need to create them manually if needed. :::

#### --subnet-config-content (string)

As an alternative to `--subnet-config-dir`, it allows specifying base64 encoded parameters for a Subnet.

## Version

#### --version (boolean)

If this is `true`, print the version and quit. Defaults to `false`.

## Advanced Options

The following options may affect the correctness of a node. Only power users should change these.

### Gossiping

#### --consensus-app-gossip-validator-size (uint)

Number of validators to gossip an AppGossip message to. Defaults to `10`.

#### --consensus-app-gossip-non-validator-size (uint)

Number of non Validators to gossip an AppGossip message to. Defaults to `0`.

#### --consensus-app-gossip-peer-size (uint)

Number of peers (which may or may not be validators) to gossip an AppGossip message to. Defaults to `0`.

#### --consensus-accepted-frontier-gossip-validator-size (uint)

Number of validators to gossip to when gossiping accepted frontier. Defaults to `0`.

#### --consensus-accepted-frontier-gossip-non-validator-size (uint)

Number of non Validators to gossip to when gossiping accepted frontier. Defaults to `0`.

```
--consensus-accepted-frontier-gossip-peer-size (uint)
Number of peers to gossip to when gossiping accepted frontier. Defaults to 15.

--consensus-accepted-frontier-gossip-frequency (duration)
Time between gossiping accepted frontiers. Defaults to 10s.

--consensus-gossip-frequency (duration)
```

This flag is deprecated as of v1.10.2. Use `--consensus-accepted-frontier-gossip-frequency` instead.

```
--consensus-on-accept-gossip-validator-size (uint)
Number of validators to gossip to each accepted container to. Defaults to 0.

--consensus-on-accept-gossip-non-validator-size (uint)
Number of nonValidators to gossip to each accepted container to. Defaults to 0.

--consensus-on-accept-gossip-peer-size (uint)
Number of peers to gossip to each accepted container to. Defaults to 10.
```

## Benchlist

```
--benchlist-duration (duration)
Maximum amount of time a peer is benchlisted after surpassing --benchlist-fail-threshold. Defaults to 15m.

--benchlist-fail-threshold (int)
Number of consecutive failed queries to a node before benching it (assuming all queries to it will fail). Defaults to 10.

--benchlist-min-failing-duration (duration)
Minimum amount of time queries to a peer must be failing before the peer is benched. Defaults to 150s.
```

## Consensus Parameters

:::note Some of these parameters can only be set on a local or private network, not on Fuji Testnet or Mainnet :::

```
--consensus-shutdown-timeout (duration)
Timeout before killing an unresponsive chain. Defaults to 5s.

--create-asset-tx-fee (int)
Transaction fee, in nAVAX, for transactions that create new assets. Defaults to 10000000 nAVAX (.01 AVAX) per transaction. This can only be changed on a local network.

--create-subnet-tx-fee (int)
Transaction fee, in nAVAX, for transactions that create new Subnets. Defaults to 1000000000 nAVAX (1 AVAX) per transaction. This can only be changed on a local network.

--create-blockchain-tx-fee (int)
Transaction fee, in nAVAX, for transactions that create new blockchains. Defaults to 1000000000 nAVAX (1 AVAX) per transaction. This can only be changed on a local network.

--transform-subnet-tx-fee (int)
Transaction fee, in nAVAX, for transactions that transform Subnets. Defaults to 1000000000 nAVAX (1 AVAX) per transaction. This can only be changed on a local network.
```

```
--add-primary-network-validator-fee (int)
Transaction fee, in nAVAX, for transactions that add new primary network validators. Defaults to 0. This can only be changed on a local network.
```

```
--add-primary-network-delegator-fee (int)
Transaction fee, in nAVAX, for transactions that add new primary network delegators. Defaults to 0. This can only be changed on a local network.
```

```
--add-subnet-validator-fee (int)
Transaction fee, in nAVAX, for transactions that add new Subnet validators. Defaults to 10000000 nAVAX (.01 AVAX).
```

```
--add-subnet-delegator-fee (int)
Transaction fee, in nAVAX, for transactions that add new Subnet delegators. Defaults to 10000000 nAVAX (.01 AVAX).
```

```
--min-delegator-stake (int)
The minimum stake, in nAVAX, that can be delegated to a validator of the Primary Network.
```

Defaults to 25000000000 (25 AVAX) on Mainnet. Defaults to 5000000 (.005 AVAX) on Test Net. This can only be changed on a local network.

**--min-delegation-fee (int)**

The minimum delegation fee that can be charged for delegation on the Primary Network, multiplied by `10,000`. Must be in the range `[0, 1000000]`. Defaults to `20000` (2%) on Mainnet. This can only be changed on a local network.

**--min-stake-duration (duration)**

Minimum staking duration. The Default on Mainnet is `336h` (two weeks). This can only be changed on a local network. This applies to both delegation and validation periods.

**--min-validator-stake (int)**

The minimum stake, in nAVAX, required to validate the Primary Network. This can only be changed on a local network.

Defaults to `2000000000000` (2,000 AVAX) on Mainnet. Defaults to `5000000` (.005 AVAX) on Test Net.

**--max-stake-duration (duration)**

The maximum staking duration, in hours. Defaults to `8760h` (365 days) on Mainnet. This can only be changed on a local network.

**--max-validator-stake (int)**

The maximum stake, in nAVAX, that can be placed on a validator on the primary network. Defaults to `3000000000000000` (3,000,000 AVAX) on Mainnet. This includes stake provided by both the validator and by delegators to the validator. This can only be changed on a local network.

**--stake-minting-period (duration)**

Consumption period of the staking function, in hours. The Default on Mainnet is `8760h` (365 days). This can only be changed on a local network.

**--stake-max-consumption-rate (uint)**

The maximum percentage of the consumption rate for the remaining token supply in the minting period, which is 1 year on Mainnet. Defaults to `120,000` which is 12% per years. This can only be changed on a local network.

**--stake-min-consumption-rate (uint)**

The minimum percentage of the consumption rate for the remaining token supply in the minting period, which is 1 year on Mainnet. Defaults to `100,000` which is 10% per years. This can only be changed on a local network.

**--stake-supply-cap (uint)**

The maximum stake supply, in nAVAX, that can be placed on a validator. Defaults to `720,000,000,000,000,000` nAVAX. This can only be changed on a local network.

**--tx-fee (int)**

The required amount of nAVAX to be burned for a transaction to be valid on the X-Chain, and for import/export transactions on the P-Chain. This parameter requires network agreement in its current form. Changing this value from the default should only be done on private networks or local network. Defaults to `1,000,000` nAVAX per transaction.

**--uptime-requirement (float)**

Fraction of time a validator must be online to receive rewards. Defaults to `0.8`. This can only be changed on a local network.

**--uptime-metric-freq (duration)**

Frequency of renewing this node's average uptime metric. Defaults to `30s`.

**Snow Parameters****--snow-concurrent-repolls (int)**

Snow consensus requires repolling transactions that are issued during low time of network usage. This parameter lets one define how aggressive the client will be in finalizing these pending transactions. This should only be changed after careful consideration of the tradeoffs of Snow consensus. The value must be at least `1` and at most `--snow-rogue-commit-threshold`. Defaults to `4`.

**--snow-sample-size (int)**

Snow consensus defines `k` as the number of validators that are sampled during each network poll. This parameter lets one define the `k` value used for consensus. This should only be changed after careful consideration of the tradeoffs of Snow consensus. The value must be at least `1`. Defaults to `20`.

**--snow-quorum-size (int)**

Snow consensus defines `alpha` as the number of validators that must prefer a transaction during each network poll to increase the confidence in the transaction. This parameter lets us define the `alpha` value used for consensus. This should only be changed after careful consideration of the tradeoffs of Snow consensus. The value must be at greater than `k/2`. Defaults to `15`.

**--snow-virtuous-commit-threshold (int)**

Snow consensus defines `beta1` as the number of consecutive polls that a virtuous transaction must increase its confidence for it to be accepted. This parameter lets us define the `beta1` value used for consensus. This should only be changed after careful consideration of the tradeoffs of Snow consensus. The value must be at least `1`. Defaults to `15`.

**--snow-rogue-commit-threshold (int)**

Snow consensus defines `beta2` as the number of consecutive polls that a rogue transaction must increase its confidence for it to be accepted. This parameter lets us define the `beta2` value used for consensus. This should only be changed after careful consideration of the tradeoffs of Snow consensus. The value must be at least `beta1`. Defaults to `20`.

```
--snow-optimal-processing (int)
Optimal number of processing items in consensus. The value must be at least 1 . Defaults to 50 .

--snow-max-processing (int)
Maximum number of processing items to be considered healthy. Reports unhealthy if more than this number of items are outstanding. The value must be at least 1 . Defaults to 1024 .

--snow-max-time-processing (duration)
Maximum amount of time an item should be processing and still be healthy. Reports unhealthy if there is an item processing for longer than this duration. The value must be greater than 0 . Defaults to 2m .
```

## ProposerVM Parameters

```
--proposervm-use-current-height (bool)
Have the ProposerVM always report the last accepted P-chain block height. Defaults to false .
```

## Continuous Profiling

You can configure your node to continuously run memory/CPU profiles and save the most recent ones. Continuous memory/CPU profiling is enabled if `--profile-continuous-enabled` is set.

```
--profile-continuous-enabled (boolean)
Whether the app should continuously produce performance profiles. Defaults to the false (not enabled).
```

```
--profile-dir (string)
If profiling enabled, node continuously runs memory/CPU profiles and puts them at this directory. Defaults to the $HOME/.avalanche/go/profiles/ .
```

```
--profile-continuous-freq (duration)
How often a new CPU/memory profile is created. Defaults to 15m .
```

```
--profile-continuous-max-files (int)
Maximum number of CPU/memory profiles files to keep. Defaults to 5.
```

## Health

```
--health-check-frequency (duration)
Health check runs with this frequency. Defaults to 30s .
```

```
--health-check-averager-halflife (duration)
Half life of averagers used in health checks (to measure the rate of message failures, for example.) Larger value --> less volatile calculation of averages. Defaults to 10s .
```

## Network

```
--network-allow-private-ips (bool)
Allows the node to connect peers with private IPs. Defaults to true .
```

```
--network-compression-enabled (bool)
If true, compress certain messages sent to peers to reduce bandwidth usage. This flag is deprecated as of v1.10.0. Use --network-compression-type instead.
```

```
--network-compression-type (string)
The type of compression to use when sending messages to peers. Defaults to gzip . Must be one of [gzip , zstd , none].
```

```
--network-initial-timeout (duration)
Initial timeout value of the adaptive timeout manager. Defaults to 5s .
```

```
--network-initial-reconnect-delay (duration)
Initial delay duration must be waited before attempting to reconnect a peer. Defaults to 1s .
```

```
--network-max-reconnect-delay (duration)
Maximum delay duration must be waited before attempting to reconnect a peer. Defaults to 1h .
```

```
--network-minimum-timeout (duration)
Minimum timeout value of the adaptive timeout manager. Defaults to 2s .
```

```
--network-maximum-timeout (duration)
Maximum timeout value of the adaptive timeout manager. Defaults to 10s .
```

```
--network-maximum-inbound-timeout (duration)
```

Maximum timeout value of an inbound message. Defines duration within which an incoming message must be fulfilled. Incoming messages containing deadline higher than this value will be overridden with this value. Defaults to `10s`.

**--network-timeout-halflife (duration)**

Half life used when calculating average network latency. Larger value --> less volatile network latency calculation. Defaults to `5m`.

**--network-timeout-coefficient (duration)**

Requests to peers will time out after [ `network-timeout-coefficient` ] \* [average request latency]. Defaults to `2`.

**--network-read-handshake-timeout (duration)**

Timeout value for reading handshake messages. Defaults to `15s`.

**--network-ping-timeout (duration)**

Timeout value for Ping-Pong with a peer. Defaults to `30s`.

**--network-ping-frequency (duration)**

Frequency of pinging other peers. Defaults to `22.5s`.

**--network-health-min-conn-peers (uint)**

Node will report unhealthy if connected to less than this many peers. Defaults to `1`.

**--network-health-max-time-since-msg-received (duration)**

Node will report unhealthy if it hasn't received a message for this amount of time. Defaults to `1m`.

**--network-health-max-time-since-msg-sent (duration)**

Network layer returns unhealthy if haven't sent a message for at least this much time. Defaults to `1m`.

**--network-health-max-portion-send-queue-full (float)**

Node will report unhealthy if its send queue is more than this portion full. Must be in [0,1]. Defaults to `0.9`.

**--network-health-max-send-fail-rate (float)**

Node will report unhealthy if more than this portion of message sends fail. Must be in [0,1]. Defaults to `0.25`.

**--network-health-max-outstanding-request-duration (duration)**

Node reports unhealthy if there has been a request outstanding for this duration. Defaults to `5m`.

**--network-max-clock-difference (duration)**

Max allowed clock difference value between this node and peers. Defaults to `1m`.

**--network-require-validator-to-connect (bool)**

If true, this node will only maintain a connection with another node if this node is a validator, the other node is a validator, or the other node is a beacon.

**--network-tcp-proxy-enabled (bool)**

Require all P2P connections to be initiated with a TCP proxy header. Defaults to `false`.

**--network-tcp-proxy-read-timeout (duration)**

Maximum duration to wait for a TCP proxy header. Defaults to `3s`.

**--network-outbound-connection-timeout (duration)**

Timeout while dialing a peer. Defaults to `30s`.

**--outbound-connection-timeout (duration)**

This flag is deprecated as of v1.10.2. Use `--network-outbound-connection-timeout` instead.

## Message Rate-Limiting

These flags govern rate-limiting of inbound and outbound messages. For more information on rate-limiting and the flags below, see package `throttling` in `AvalancheGo`.

### CPU Based

Rate-limiting based on how much CPU usage a peer causes.

**--throttler-inbound-cpu-validator-alloc (float)**

Number of CPU allocated for use by validators. Value should be in range (0, total core count]. Defaults to half of the number of CPUs on the machine.

**--throttler-inbound-cpu-max-recheck-delay (duration)**

In the CPU rate-limiter, check at least this often whether the node's CPU usage has fallen to an acceptable level. Defaults to `5s`.

**--throttler-inbound-disk-max-recheck-delay (duration)**

In the disk-based network throttler, check at least this often whether the node's disk usage has fallen to an acceptable level. Defaults to `5s`.

#### `--throttler-inbound-cpu-max-non-validator-usage (float)`

Number of CPUs that if fully utilized, will rate limit all non-validators. Value should be in range [0, total core count]. Defaults to %80 of the number of CPUs on the machine.

#### `--throttler-inbound-cpu-max-non-validator-node-usage (float)`

Maximum number of CPUs that a non-validator can utilize. Value should be in range [0, total core count]. Defaults to the number of CPUs / 8.

#### `--throttler-inbound-disk-validator-alloc (float)`

Maximum number of disk reads/writes per second to allocate for use by validators. Must be > 0. Defaults to `1000 GiB/s`.

#### `--throttler-inbound-disk-max-non-validator-usage (float)`

Number of disk reads/writes per second that, if fully utilized, will rate limit all non-validators. Must be >= 0. Defaults to `1000 GiB/s`.

#### `--throttler-inbound-disk-max-non-validator-node-usage (float)`

Maximum number of disk reads/writes per second that a non-validator can utilize. Must be >= 0. Defaults to `1000 GiB/s`.

### Bandwidth Based

Rate-limiting based on the bandwidth a peer uses.

#### `--throttler-inbound-bandwidth-refill-rate (uint)`

Max average inbound bandwidth usage of a peer, in bytes per second. See interface `throttling.BandwidthThrottler`. Defaults to `512`.

#### `--throttler-inbound-bandwidth-max-burst-size (uint)`

Max inbound bandwidth a node can use at once. See interface `throttling.BandwidthThrottler`. Defaults to `2 MiB`.

### Message Size Based

Rate-limiting based on the total size, in bytes, of unprocessed messages.

#### `--throttler-inbound-at-large-alloc-size (uint)`

Size, in bytes, of at-large allocation in the inbound message throttler. Defaults to `6291456` (6 MiB).

#### `--throttler-inbound-validator-alloc-size (uint)`

Size, in bytes, of validator allocation in the inbound message throttler. Defaults to `33554432` (32 MiB).

#### `--throttler-inbound-node-max-at-large-bytes (uint)`

Maximum number of bytes a node can take from the at-large allocation of the inbound message throttler. Defaults to `2097152` (2 MiB).

### Message Based

Rate-limiting based on the number of unprocessed messages.

#### `--throttler-inbound-node-max-processing-msgs (uint)`

Node will stop reading messages from a peer when it is processing this many messages from the peer. Will resume reading messages from the peer when it is processing less than this many messages. Defaults to `1024`.

### Outbound

Rate-limiting for outbound messages.

#### `--throttler-outbound-at-large-alloc-size (uint)`

Size, in bytes, of at-large allocation in the outbound message throttler. Defaults to `33554432` (32 MiB).

#### `--throttler-outbound-validator-alloc-size (uint)`

Size, in bytes, of validator allocation in the outbound message throttler. Defaults to `33554432` (32 MiB).

#### `--throttler-outbound-node-max-at-large-bytes (uint)`

Maximum number of bytes a node can take from the at-large allocation of the outbound message throttler. Defaults to `2097152` (2 MiB).

### Connection Rate-Limiting

#### `--network-inbound-connection-throttling-cooldown (duration)`

Node will upgrade an inbound connection from a given IP at most once within this duration. Defaults to `10s`. If 0 or negative, will not consider recency of last upgrade when deciding whether to upgrade.

#### `--inbound-connection-throttling-cooldown (duration)`

This flag is deprecated as of v1.10.2. Use `--network-inbound-connection-throttling-cooldown` instead.

#### `--network-inbound-connection-throttling-max-conns-per-sec (uint)`

Node will accept at most this many inbound connections per second. Defaults to `512`.

#### `--inbound-connection-throttling-max-conns-per-sec (uint)`

This flag is deprecated as of v1.10.2. Use `--network-inbound-connection-throttling-max-conns-per-sec` instead.

#### `--network-outbound-connection-throttling-rps (uint)`

Node makes at most this many outgoing peer connection attempts per second. Defaults to 50 .

--outbound-connection-throttling-rps (uint)

This flag is deprecated as of v1.10.2. Use --network-outbound-connection-throttling-rps instead.

### Peer List Gossiping

Nodes gossip peers to each other so that each node can have an up-to-date peer list. A node gossips --network-peer-list-num-validator-ips validator IPs to --network-peer-list-validator-gossip-size validators, --network-peer-list-non-validator-gossip-size nonValidators and --network-peer-list-peers-gossip-size peers every --network-peer-list-gossip-frequency .

--network-peer-list-num-validator-ips (int)

Number of validator IPs to gossip to other nodes Defaults to 15 .

--network-peer-list-validator-gossip-size (int)

Number of validators that the node will gossip peer list to. Defaults to 20 .

--network-peer-list-non-validator-gossip-size (int)

Number of non-validators that the node will gossip peer list to. Defaults to 0 .

--network-peer-list-peers-gossip-size (int)

Number of total peers (including non-validator or validator) that the node will gossip peer list to Defaults to 0 .

--network-peer-list-gossip-frequency (duration)

Frequency to gossip peers to other nodes. Defaults to 1m .

--network-peer-read-buffer-size (int)

Size of the buffer that peer messages are read into (there is one buffer per peer), defaults to 8 KiB (8192 Bytes).

--network-peer-write-buffer-size (int)

Size of the buffer that peer messages are written into (there is one buffer per peer), defaults to 8 KiB (8192 Bytes).

### Resource Usage Tracking

--meter-vm-enabled (bool)

Enable Meter VMs to track VM performance with more granularity. Defaults to true .

--system-tracker-frequency (duration)

Frequency to check the real system usage of tracked processes. More frequent checks --> usage metrics are more accurate, but more expensive to track. Defaults to 500ms .

--system-tracker-processing-halflife (duration)

Half life to use for the processing requests tracker. Larger half life --> usage metrics change more slowly. Defaults to 15s .

--system-tracker-cpu-halflife (duration)

Half life to use for the CPU tracker. Larger half life --> CPU usage metrics change more slowly. Defaults to 15s .

--system-tracker-disk-halflife (duration)

Half life to use for the disk tracker. Larger half life --> disk usage metrics change more slowly. Defaults to 1m .

--system-tracker-disk-required-available-space (uint)

"Minimum number of available bytes on disk, under which the node will shutdown. Defaults to 536870912 (512 MiB).

--system-tracker-disk-warning-threshold-available-space (uint)

Warning threshold for the number of available bytes on disk, under which the node will be considered unhealthy. Must be >= --system-tracker-disk-required-available-space . Defaults to 1073741824 (1 GiB).

### Plugins

--plugin-dir (string)

Sets the directory for [VM plugins](#). The default value is \$HOME/.avalanchego/plugins .

### Virtual Machine (VM) Configs

--vm-aliases-file (string)

Path to JSON file that defines aliases for Virtual Machine IDs. Defaults to ~/.avalanchego/configs/vms/aliases.json . This flag is ignored if --vm-aliases-file-content is specified. Example content:

```
{
 "tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH": [
 "timestampvpm",
 "timerpc"
]
}
```

The above example aliases the VM whose ID is `"tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH"` to `"timestampvpm"` and `"timerpc"`.

`--vm-aliases-file-content (string)`

As an alternative to `--vm-aliases-file`, it allows specifying base64 encoded aliases for Virtual Machine IDs.

## Indexing

`--index-allow-incomplete (boolean)`

If true, allow running the node in such a way that could cause an index to miss transactions. Ignored if index is disabled. Defaults to `false`.

## Router

`--router-health-max-drop-rate (float)`

Node reports unhealthy if the router drops more than this portion of messages. Defaults to `1`.

`--router-health-max-outstanding-requests (uint)`

**Node reports unhealthy if there are more than this many outstanding consensus requests (Get, PullQuery, etc.) over all chains. Defaults to 1024.**

**sidebar\_position: 5 description: Reference for all available chain config options and flags.**

## Chain Configs

Some chains allow the node operator to provide a custom configuration. AvalancheGo can read chain configurations from files and pass them to the corresponding chains on initialization.

AvalancheGo looks for these files in the directory specified by `--chain-config-dir` AvalancheGo flag, as documented [here](#). If omitted, value defaults to `$HOME/.avalanchego/configs/chains`. This directory can have sub-directories whose names are chain IDs or chain aliases. Each sub-directory contains the configuration for the chain specified in the directory name. Each sub-directory should contain a file named `config`, whose value is passed in when the corresponding chain is initialized (see below for extension). For example, config for the C-Chain should be at: `(chain-config-dir)/C/config.json`.

This also applies to Subnets, for example, if a Subnet's chain id is `2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQzTFhBq8qeqh6tt`, the config for this chain should be at `(chain-config-dir)/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQzTFhBq8qeqh6tt/config.json`

:::tip

By default, none of these directories and/or files exist. You would need to create them manually if needed.

:::

The filename extension that these files should have, and the contents of these files, is VM-dependent. For example, some chains may expect `config.txt` while others expect `config.json`. If multiple files are provided with the same name but different extensions (for example `config.json` and `config.txt`) in the same sub-directory, AvalancheGo will exit with an error.

For a given chain, AvalancheGo will follow the sequence below to look for its config file, where all folder and file names are case sensitive:

- First it looks for a config sub-directory whose name is the chain ID - If it isn't found, it looks for a config sub-directory whose name is the chain's primary alias - If it's not found, it looks for a config sub-directory whose name is another alias for the chain

Alternatively, for some setups it might be more convenient to provide config entirely via the command line. For that, you can use AvalancheGo `--chain-config-content` flag, as documented [here](#).

It is not required to provide these custom configurations. If they are not provided, a VM-specific default config will be used. And the values of these default config are printed when the node starts.

## C-Chain Configs

In order to specify a config for the C-Chain, a JSON config file should be placed at `(chain-config-dir)/C/config.json`. This file does not exist by default.

For example if `chain-config-dir` has the default value which is `$HOME/.avalanchego/configs/chains`, then `config.json` should be placed at `$HOME/.avalanchego/configs/chains/C/config.json`.

The C-Chain config is printed out in the log when a node starts. Default values for each config flag are specified below.

Default values are overridden only if specified in the given config file. It is recommended to only provide values which are different from the default, as that makes the config more resilient to future default changes. Otherwise, if defaults change, your node will remain with the old values, which might adversely affect your node operation.

## State Sync

**state-sync-enabled (boolean)**

Set to `true` to start the chain with state sync enabled. The peer will download chain state from peers up to a recent block near tip, then proceed with normal bootstrapping.

Defaults to perform state sync if starting a new node from scratch. However, if running with an existing database it will default to false and not perform state sync on subsequent runs.

Please note that if you need historical data, state sync isn't the right option. However, it is sufficient if you are just running a validator.

**state-sync-skip-resume (boolean)**

If set to `true`, the chain will not resume a previously started state sync operation that did not complete. Normally, the chain should be able to resume state syncing without any issue. Defaults to `false`.

**state-sync-min-blocks (int)**

Minimum number of blocks the chain should be ahead of the local node to prefer state syncing over bootstrapping. If the node's database is already close to the chain's tip, bootstrapping is more efficient. Defaults to `300000`.

**state-sync-ids (string)**

Comma separated list of node IDs (prefixed with `NodeID-`) to fetch state sync data from. An example setting of this field would be `--state-sync-ids="NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg,NodeID-MFrZFVCXPv5iCn6M9K6XduxGTYp891xxZ"`. If not specified (or empty), peers are selected at random. Defaults to empty string ( `"` ).

**state-sync-server-trie-cache (int)**

Size of trie cache used for providing state sync data to peers in MBs. Should be a multiple of `64`. Defaults to `64`.

**Continuous Profiling****continuous-profiler-dir (string)**

Enables the continuous profiler (captures a CPU/Memory/Lock profile at a specified interval). Defaults to `" "`. If a non-empty string is provided, it enables the continuous profiler and specifies the directory to place the profiles in.

**continuous-profiler-frequency (duration)**

Specifies the frequency to run the continuous profiler. Defaults `900000000000` nano seconds which is 15 minutes.

**continuous-profiler-max-files (int)**

Specifies the maximum number of profiles to keep before removing the oldest. Defaults to `5`.

**Enabling Avalanche Specific APIs****snowman-api-enabled (boolean)**

Enables the Snowman API. Defaults to `false`.

**coreth-admin-api-enabled (boolean)**

Enables the Admin API. Defaults to `false`.

**coreth-admin-api-dir (string)**

Specifies the directory for the Admin API to use to store CPU/Mem/Lock Profiles. Defaults to `" "`.

**Enabling EVM APIs****eth-apis ([]string)**

Use the `eth-apis` field to specify the exact set of below services to enable on your node. If this field is not set, then the default list will be: `["eth", "eth-filter", "net", "web3", "internal-eth", "internal-blockchain", "internal-transaction"]`.

:::note

The names used in this configuration flag have been updated in Coreth v0.8.14. The previous names containing `public-` and `private-` are deprecated. While the current version continues to accept deprecated values, they may not be supported in future updates and updating to the new values is recommended.

The mapping of deprecated values and their updated equivalent follows:

| Deprecated                       | Use instead               |
|----------------------------------|---------------------------|
| <code>public-eth</code>          | <code>eth</code>          |
| <code>public-eth-filter</code>   | <code>eth-filter</code>   |
| <code>private-admin</code>       | <code>admin</code>        |
| <code>private-debug</code>       | <code>debug</code>        |
| <code>public-debug</code>        | <code>debug</code>        |
| <code>internal-public-eth</code> | <code>internal-eth</code> |

|                                  |                      |
|----------------------------------|----------------------|
| internal-public-blockchain       | internal-blockchain  |
| internal-public-transaction-pool | internal-transaction |
| internal-public-tx-pool          | internal-tx-pool     |
| internal-public-debug            | internal-debug       |
| internal-private-debug           | internal-debug       |
| internal-public-account          | internal-account     |
| internal-private-personal        | internal-personal    |

:::

#### :::note

If you populate this field, it will override the defaults so you must include every service you wish to enable.

:::

#### eth

The API name `public-eth` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `eth`.

Adds the following RPC calls to the `eth_*` namespace. Defaults to `true`.

`eth_coinbase` `eth_etherbase`

#### eth-filter

The API name `public-eth-filter` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `eth-filter`.

Enables the public filter API for the `eth_*` namespace. Defaults to `true`.

Adds the following RPC calls (see [here](#) for complete documentation):

- `eth_newPendingTransactionFilter`
- `eth_newPendingTransactions`
- `eth_newAcceptedTransactions`
- `eth_newBlockFilter`
- `eth_newHeads`
- `eth_logs`
- `eth_newFilter`
- `eth_getLogs`
- `eth_uninstallFilter`
- `eth_getFilterLogs`
- `eth_getFilterChanges`

#### admin

The API name `private-admin` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `admin`.

Adds the following RPC calls to the `admin_*` namespace. Defaults to `false`.

- `admin_importChain`
- `admin_exportChain`

#### debug

The API names `private-debug` and `public-debug` are deprecated as of v1.7.15, and the APIs previously under these names have been migrated to `debug`.

Adds the following RPC calls to the `debug_*` namespace. Defaults to `false`.

- `debug_dumpBlock`
- `debug_accountRange`
- `debug_preimage`
- `debug_getBadBlocks`
- `debug_storageRangeAt`
- `debug_getModifiedAccountsByNumber`
- `debug_getModifiedAccountsByHash`
- `debug_getAccessibleState`

#### net

Adds the following RPC calls to the `net_*` namespace. Defaults to `true`.

- `net_listening`
- `net_peerCount`
- `net_version`

Note: Coreth is a virtual machine and does not have direct access to the networking layer, so `net_listening` always returns true and `net_peerCount` always returns 0. For accurate metrics on the network layer, users should use the AvalancheGo APIs.

#### **debug-tracer**

Adds the following RPC calls to the `debug_*` namespace. Defaults to `false`.

- `debug_traceChain`
- `debug_traceBlockByNumber`
- `debug_traceBlockByHash`
- `debug_traceBlock`
- `debug_traceBadBlock`
- `debug_intermediateRoots`
- `debug_traceTransaction`
- `debug_traceCall`

#### **web3**

Adds the following RPC calls to the `web3_*` namespace. Defaults to `true`.

- `web3_clientVersion`
- `web3_sha3`

#### **internal-eth**

The API name `internal-public-eth` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-eth`.

Adds the following RPC calls to the `eth_*` namespace. Defaults to `true`.

- `eth_gasPrice`
- `eth_baseFee`
- `eth_maxPriorityFeePerGas`
- `eth_feeHistory`

#### **internal-blockchain**

The API name `internal-public-blockchain` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-blockchain`.

Adds the following RPC calls to the `eth_*` namespace. Defaults to `true`.

- `eth_chainId`
- `eth_blockNumber`
- `eth_getBalance`
- `eth_getAssetBalance`
- `eth_getProof`
- `eth_getHeaderByNumber`
- `eth_getHeaderByHash`
- `eth_getBlockByNumber`
- `eth_getBlockByHash`
- `eth_getUncleBlockByNumberAndIndex`
- `eth_getUncleBlockByBlockHashAndIndex`
- `eth_getUncleCountByBlockNumber`
- `eth_getUncleCountByBlockHash`
- `eth_getCode`
- `eth_getStorageAt`
- `eth_call`
- `eth_estimateGas`
- `eth_createAccessList`

#### **internal-transaction**

The API name `internal-public-transaction-pool` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-transaction`.

Adds the following RPC calls to the `eth_*` namespace. Defaults to `true`.

- `eth_getBlockTransactionCountByNumber`
- `eth_getBlockTransactionCountByHash`
- `eth_getTransactionByBlockNumberAndIndex`
- `eth_getTransactionByBlockHashAndIndex`
- `eth_getRawTransactionByBlockNumberAndIndex`
- `eth_getRawTransactionByBlockHashAndIndex`
- `eth_getTransactionCount`
- `eth_getTransactionByHash`
- `eth_getRawTransactionByHash`
- `eth_getTransactionReceipt`
- `eth_sendTransaction`
- `eth_fillTransaction`
- `eth_sendRawTransaction`
- `eth_sign`

- `eth_signTransaction`
- `eth_pendingTransactions`
- `eth_resend`

#### **internal-tx-pool**

The API name `internal-public-tx-pool` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-tx-pool`.

Adds the following RPC calls to the `txpool_*` namespace. Defaults to `false`.

- `txpool_content`
- `txpool_contentFrom`
- `txpool_status`
- `txpool_inspect`

#### **internal-debug**

The API names `internal-private-debug` and `internal-public-debug` are deprecated as of v1.7.15, and the APIs previously under these names have been migrated to `internal-debug`.

Adds the following RPC calls to the `debug_*` namespace. Defaults to `false`.

- `debug_getHeaderRlp`
- `debug_getBlockRlp`
- `debug_printBlock`
- `debug_chainedbProperty`
- `debug_chainedbCompact`

#### **debug-handler**

Adds the following RPC calls to the `debug_*` namespace. Defaults to `false`.

- `debug_verbosity`
- `debug_vmodule`
- `debug_backtraceAt`
- `debug_memStats`
- `debug_gcStats`
- `debug_blockProfile`
- `debug_setBlockProfileRate`
- `debug_writeBlockProfile`
- `debug_mutexProfile`
- `debug_setMutexProfileFraction`
- `debug_writeMutexProfile`
- `debug_writeMemProfile`
- `debug_stacks`
- `debug_freeOSMemory`
- `debug_setGCPercen`

#### **internal-account**

The API name `internal-public-account` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-account`.

Adds the following RPC calls to the `eth_*` namespace. Defaults to `true`.

- `eth_accounts`

#### **internal-personal**

The API name `internal-private-personal` is deprecated as of v1.7.15, and the APIs previously under this name have been migrated to `internal-personal`.

Adds the following RPC calls to the `personal_*` namespace. Defaults to `false`.

- `personal_listAccounts`
- `personal_listWallets`
- `personal_openWallet`
- `personal_deriveAccount`
- `personal_newAccount`
- `personal_importRawKey`
- `personal_unlockAccount`
- `personal_lockAccount`
- `personal_sendTransaction`
- `personal_signTransaction`
- `personal_sign`
- `personal_ecRecover`
- `personal_signAndSendTransaction`
- `personal_initializeWallet`
- `personal_unpair`

## **API Configuration**

**rpc-gas-cap (int)**

The maximum gas to be consumed by an RPC Call (used in `eth_estimateGas` and `eth_call`). Defaults to `50,000,000`.

**rpc-tx-fee-cap (int)**

Global transaction fee (`price * gaslimit`) cap (measured in AVAX) for send-transaction variants. Defaults to `100`.

**api-max-duration (duration)**

Maximum API call duration. If API calls exceed this duration, they will time out. Defaults to `0` (no maximum).

**api-max-blocks-per-request (int)**

Maximum number of blocks to serve per `getLogs` request. Defaults to `0` (no maximum).

**ws-cpu-refill-rate (duration)**

The refill rate specifies the maximum amount of CPU time to allot a single connection per second. Defaults to no maximum (`0`).

**ws-cpu-max-stored (duration)**

Specifies the maximum amount of CPU time that can be stored for a single WS connection. Defaults to no maximum (`0`).

**allow-unfinalized-queries (boolean)**

Allows queries for unfinalized (not yet accepted) blocks/transactions. Defaults to `false`.

**accepted-cache-size (int)**

Specifies the depth to keep accepted headers and accepted logs in the cache. This is particularly useful to improve the performance of `eth_getLogs` for recent logs.

## Transaction Pool

**local-txs-enabled (boolean)**

Enables local transaction handling (prioritizes transactions submitted through this node). Defaults to `false`.

**allow-unprotected-txs (boolean)**

If `true`, the APIs will allow transactions that are not replay protected (EIP-155) to be issued through this node. Defaults to `false`.

**allow-unprotected-tx-hashes ([]TxHash)**

Specifies an array of transaction hashes that should be allowed to bypass replay protection. This flag is intended for node operators that want to explicitly allow specific transactions to be issued through their API. Defaults to an empty list.

**remote-tx-gossip-only-enabled (boolean)**

If `true`, the node will only gossip remote transactions to prevent transactions issued through this node from being broadcast to the network. Defaults to `false`.

**tx-regossip-frequency (duration)**

Amount of time that should elapse before we attempt to re-gossip a transaction that was already gossiped once. Defaults to `60000000000` nano seconds which is 1 minute.

**tx-regossip-max-size (int)**

Maximum number of transactions to re-gossip at once. Defaults to `15`.

**tx-pool-journal (string)**

Specifies file path to a transaction journal to store local transactions that survive between node restarts.

Defaults to empty string and being disabled. To enable transaction pool journaling, the user must specify a non-empty journal and enable local transactions via `local-txs-enabled`.

**tx-pool-rejournal (duration)**

Time interval to regenerate the local transaction journal. Defaults to 1 hour.

**tx-pool-price-limit (int)**

Minimum gas price to enforce for acceptance into the pool. Defaults to 1 wei.

**tx-pool-price-bump (int)**

Minimum price bump percentage to replace an already existing transaction (nonce). Defaults to 10%.

**tx-pool-account-slots (int)**

Number of executable transaction slots guaranteed per account. Defaults to 16.

**tx-pool-global-slots (int)**

Maximum number of executable transaction slots for all accounts. Defaults to 5120.

**tx-pool-account-queue (int)**

Maximum number of non-executable transaction slots permitted per account. Defaults to 64.

**tx-pool-global-queue (int)**

Maximum number of non-executable transaction slots for all accounts. Defaults to 1024.

**Metrics****metrics-enabled (boolean)**

Enables metrics. Defaults to `false`.

**metrics-expensive-enabled (boolean)**

Enables expensive metrics. Defaults to `false`.

**Snapshots****snapshot-async (boolean)**

If `true`, allows snapshot generation to be executed asynchronously. Defaults to `true`.

**snapshot-verification-enabled (boolean)**

If `true`, verifies the complete snapshot after it has been generated. Defaults to `false`.

**Logging****log-level (string)**

Defines the log level for the chain. Must be one of `"trace"`, `"debug"`, `"info"`, `"warn"`, `"error"`, `"crit"`. Defaults to `"info"`.

**log-json-format (bool)**

If `true`, changes logs to JSON format. Defaults to `false`.

**Keystore Settings****keystore-directory (string)**

The directory that contains private keys. Can be given as a relative path. If empty, uses a temporary directory at `coreth-keystore`. Defaults to the empty string (`""`).

**keystore-external-signer (string)**

Specifies an external URI for a clef-type signer. Defaults to the empty string (`""` as not enabled).

**keystore-insecure-unlock-allowed (bool)**

If `true`, allow users to unlock accounts in unsafe HTTP environment. Defaults to `false`.

**Database****trie-clean-cache (int)**

Size of cache used for clean trie nodes (in MBs). Should be a multiple of `64`. Defaults to `512`.

**trie-dirty-cache (int)**

Size of cache used for dirty trie nodes (in MBs). When the dirty nodes exceed this limit, they are written to disk. Defaults to `256`.

**trie-dirty-commit-target (int)**

Memory limit to target in the dirty cache before performing a commit (in MBs). Defaults to `20`.

**snapshot-cache (int)**

Size of the snapshot disk layer clean cache (in MBs). Should be a multiple of `64`. Defaults to `256`.

**trie-clean-journal (string)**

Directory to use to save the trie clean cache (must be populated to enable journaling the trie clean cache). Empty and disabled by default.

**trie-clean-rejournal (duration)**

Frequency to re-journal the trie clean cache to disk (minimum 1 minute, must be populated to enable journaling the trie clean cache).

**acceptor-queue-limit (int)**

Specifies the maximum number of blocks to queue during block acceptance before blocking on Accept. Defaults to `64`.

**commit-interval (int)**

Specifies the commit interval at which to persist the merkle trie to disk. Defaults to `4096`.

**pruning-enabled (boolean)**

If `true`, database pruning of obsolete historical data will be enabled. This reduces the amount of data written to disk, but does not delete any state that is written to the disk previously. This flag should be set to `false` for nodes that need access to all data at historical roots. Pruning will be done only for new data. Defaults to `false` in v1.4.9, and `true` in subsequent versions.

:::note

If a node is ever run with `pruning-enabled` as `false` (archival mode), setting `pruning-enabled` to `true` will result in a warning and the node will shut down. This is to protect against unintentional misconfigurations of an archival node.

To override this and switch to pruning mode, in addition to `pruning-enabled: true`, `allow-missing-tries` should be set to `true` as well.

:::

#### `populate-missing-tries (*uint64)`

If non-nil, sets the starting point for repopulating missing tries to re-generate archival merkle forest.

To restore an archival merkle forest that has been corrupted (missing trie nodes for a section of the blockchain), specify the starting point of the last block on disk, where the full trie was available at that block to re-process blocks from that height onwards and re-generate the archival merkle forest on startup. This flag should be used once to re-generate the archival merkle forest and should be removed from the config after completion. This flag will cause the node to delay starting up while it re-processes old blocks.

#### `populate-missing-tries-parallelism (int)`

Number of concurrent readers to use when re-populating missing tries on startup. Defaults to 1024.

#### `allow-missing-tries (boolean)`

If `true`, allows a node that was once configured as archival to switch to pruning mode. Defaults to `false`.

#### `preimages-enabled (boolean)`

If `true`, enables preimages. Defaults to `false`.

#### `offline-pruning-enabled (boolean)`

If `true`, offline pruning will run on startup and block until it completes (approximately one hour on Mainnet). This will reduce the size of the database by deleting old trie nodes. **While performing offline pruning, your node will not be able to process blocks and will be considered offline.** While ongoing, the pruning process consumes a small amount of additional disk space (for deletion markers and the bloom filter). For more information see [here](#).

Since offline pruning deletes old state data, this should not be run on nodes that need to support archival API requests.

This is meant to be run manually, so after running with this flag once, it must be toggled back to false before running the node again. Therefore, you should run with this flag set to true and then set it to false on the subsequent run.

#### `offline-pruning-bloom-filter-size (int)`

This flag sets the size of the bloom filter to use in offline pruning (denominated in MB and defaulting to 512 MB). The bloom filter is kept in memory for efficient checks during pruning and is also written to disk to allow pruning to resume without re-generating the bloom filter.

The active state is added to the bloom filter before iterating the DB to find trie nodes that can be safely deleted, any trie nodes not in the bloom filter are considered safe for deletion. The size of the bloom filter may impact its false positive rate, which can impact the results of offline pruning. This is an advanced parameter that has been tuned to 512 MB and should not be changed without thoughtful consideration.

#### `offline-pruning-data-directory (string)`

This flag must be set when offline pruning is enabled and sets the directory that offline pruning will use to write its bloom filter to disk. This directory should not be changed in between runs until offline pruning has completed.

#### `tx-lookup-limit (uint64)`

Number of recent blocks for which to maintain transaction lookup indices in the database. If set to 0, transaction lookup indices will be maintained for all blocks. Defaults to 0.

## VM Networking

#### `max-outbound-active-requests (int)`

Specifies the maximum number of outbound VM2VM requests in flight at once. Defaults to 16.

#### `max-outbound-active-cross-chain-requests (int)`

Specifies the maximum number of outbound cross-chain requests in flight at once. Defaults to 64.

## Miscellaneous

#### `airdrop (string)`

Path to a json file that contains a list of addresses for a genesis airdrop. Each address will be airdropped `AirdropAmount` at genesis, and the hash of the airdrop file must match `AirdropHash`. `AirdropAmount` and `AirdropHash` are part of the genesis config. This option applies to `subnet-evm` only (not applicable to `coreth`).

#### `skip-upgrade-check (bool)`

If set to `true`, the chain will skip verifying that all expected network upgrades have taken place before the last accepted block on startup. This allows node operators to recover if their node has accepted blocks after a network upgrade with a version of the code prior to the upgrade. Defaults to `false`.

## X-Chain Configs

In order to specify a config for the X-Chain, a JSON config file should be placed at `{chain-config-dir}/X/config.json`.

For example if `chain-config-dir` has the default value which is `$HOME/.avalanchego/configs/chains`, then `config.json` can be placed at `$HOME/.avalanchego/configs/chains/X/config.json`.

This allows you to specify a config to be passed into the X-Chain. The default values for this config are:

```
{
 "index-transactions": false,
 "index-allow-incomplete": false
}
```

Default values are overridden only if explicitly specified in the config.

The parameters are as follows:

### Transaction Indexing

#### `index-transactions` (boolean)

Enables AVM transaction indexing if set to `true`. Default value is `false`. When set to `true`, AVM transactions are indexed against the `address` and `assetID` involved. This data is available via [avm.getAddressTxs API](#).

:::note If `index-transactions` is set to `true`, it must always be set to `true` for the node's lifetime. If set to `false` after having been set to `true`, the node will refuse to start unless `index-allow-incomplete` is also set to `true` (see below). :::

#### `index-allow-incomplete` (boolean)

Allows incomplete indices. Default value is `false`.

This config value is ignored if there is no X-Chain indexed data in the DB and `index-transactions` is set to `false`.

## Subnet Chain Configs

As mentioned above, if a Subnet's chain id is `2ebCneCbwtbjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt`, the config for this chain should be at `{chain-config-dir}/2ebCneCbwtbjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt/config.json`

## FAQ

- When using `getBlockNumber` it will return finalized blocks. To allow for queries for unfinalized (not yet accepted) blocks/transactions use `allow-unfinalized-queries` and set to `true` (by default it is set to `false`)
- When deactivating offline pruning (`pruning-enabled: false`) from previously enabled state, this will not impact blocks whose state was already pruned. This will return missing trie node errors, as the node can't lookup the state of a historical block if that state was deleted.

---

**sidebar\_position: 1**

## Node Backup and Restore

Once you have your node up and running, it's time to prepare for disaster recovery. Should your machine ever have a catastrophic failure due to either hardware or software issues, or even a case of natural disaster, it's best to be prepared for such a situation by making a backup.

When running, a complete node installation along with the database can grow to be multiple gigabytes in size. Having to back up and restore such a large volume of data can be expensive, complicated and time-consuming. Luckily, there is a better way.

Instead of having to back up and restore everything, we need to back up only what is essential, that is, those files that cannot be reconstructed because they are unique to your node. For AvalancheGo node, unique files are those that identify your node on the network, in other words, files that define your NodeID.

Even if your node is a validator on the network and has multiple delegations on it, you don't need to worry about backing up anything else, because the validation and delegation transactions are also stored on the blockchain and will be restored during bootstrapping, along with the rest of the blockchain data.

The installation itself can be easily recreated by installing the node on a new machine, and all the remaining gigabytes of blockchain data can be easily recreated by the process of bootstrapping, which copies the data over from other network peers. However, if you would like to speed up the process, see the [Database Backup and Restore section](#)

### NodeID

:::warning

If more than one running nodes share the same NodeID, the communications from other nodes in the Avalanche network to this NodeID will be random to one of these nodes. If this NodeID is of a validator, it will dramatically impact the uptime calculation of the validator which will very likely disqualify the validator from receiving the staking rewards. Please make sure only one node with the same NodeID run at one time.

:::

NodeID is a unique identifier that differentiates your node from all the other peers on the network. It's a string formatted like `NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD`. You can look up the technical background of how the NodeID is constructed [here](#). In essence, NodeID is defined by two files:

- `staker.crt`
- `staker.key`

In the default installation, they can be found in the working directory, specifically in `~/.avalanchego/staking/`. All we need to do to recreate the node on another machine is to run a new installation with those same two files. If these two files are removed from a node, which is restarted afterwards, they will be recreated and a new node ID will be assigned.

:::caution

If you have users defined in the keystore of your node, then you need to back up and restore those as well. [Keystore API](#) has methods that can be used to export and import user keys. Note that Keystore API is used by developers only and not intended for use in production nodes. If you don't know what a keystore API is and have not used it, you don't need to worry about it.

:::

## Backup

To back up your node, we need to store `staker.crt` and `staker.key` files somewhere safe and private, preferably to a different computer, to your private storage in the cloud, a USB stick or similar. Storing them to a couple of different, secure locations increases the safety.

:::caution

If someone gets a hold of your staker files, they still cannot get to your funds, as they are controlled by the wallet private keys, not by the node. But, they could re-create your node somewhere else, and depending on the circumstances make you lose the staking rewards. So make sure your staker files are secure.

:::

Let's get the staker files off the machine running the node.

### From Local Node

If you're running the node locally, on your desktop computer, just navigate to where the files are and copy them somewhere safe.

On a default Linux installation, the path to them will be `/home/USERNAME/.avalanchego/staking/`, where `USERNAME` needs to be replaced with the actual username running the node. Select and copy the files from there to a backup location. You don't need to stop the node to do that.

### From Remote Node Using `scp`

`scp` is a 'secure copy' command line program, available built-in on Linux and MacOS computers. There is also a Windows version, `pscp`, as part of the [PuTTY](#) package. If using `pscp`, in the following commands replace each usage of `scp` with `pscp -scp`.

To copy the files from the node, you will need to be able to remotely log into the machine. You can use account password, but the secure and recommended way is to use the SSH keys. The procedure for acquiring and setting up SSH keys is highly dependent on your cloud provider and machine configuration. You can refer to our [Amazon Web Services](#) and [Microsoft Azure](#) setup guides for those providers. Other providers will have similar procedures.

When you have means of remote login into the machine, you can copy the files over with the following command:

```
scp -r ubuntu@PUBLICIP:/home/ubuntu/.avalanchego/staking ~/avalanche_backup
```

This assumes the username on the machine is `ubuntu`, replace with correct username in both places if it is different. Also, replace `PUBLICIP` with the actual public IP of the machine. If `scp` doesn't automatically use your downloaded SSH key, you can point to it manually:

```
scp -i /path/to/the/key.pem -r ubuntu@PUBLICIP:/home/ubuntu/.avalanchego/staking ~/avalanche_backup
```

Once executed, this command will create `avalanche_backup` directory in your home directory and place staker files in it. You need to store them somewhere safe.

## Restore

To restore your node from a backup, we need to do the reverse: restore `staker.key` and `staker.crt` from the backup to the working directory of the node.

First, we need to do the usual [installation](#) of the node. This will create a new NodeID, which we need to replace. When the node is installed correctly, log into the machine where the node is running and stop it:

```
sudo systemctl stop avalanchego
```

We're ready to restore the node.

### To Local Node

If you're running the node locally, just copy the `staker.key` and `staker.crt` files from the backup location into the working directory, which on the default Linux installation will be `/home/USERNAME/.avalanchego/staking/`. Replace `USERNAME` with the actual username used to run the node.

### To Remote Node Using `scp`

Again, the process is just the reverse operation. Using `scp` we need to copy the `staker.key` and `staker.crt` files from the backup location into the remote working directory. Assuming the backed up files are located in the directory where the above backup procedure placed them:

```
scp ~/avalanche_backup/staker.* ubuntu@PUBLICIP:/home/ubuntu/.avalanchego/staking
```

Or if you need to specify the path to the SSH key:

```
scp -i /path/to/the/key.pem ~/avalanche_backup/staker.* ubuntu@PUBLICIP:/home/ubuntu/.avalanchego/staking
```

And again, replace `ubuntu` with correct username if different, and `PUBLICIP` with the actual public IP of the machine running the node, as well as the path to the SSH key if used.

#### Restart the Node and Verify

Once the files have been replaced, log into the machine and start the node using:

```
sudo systemctl start avalanchego
```

You can now check that the node is restored with the correct NodeID by issuing the [getNodeID](#) API call in the same console you ran the previous command:

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

You should see your original NodeID. Restore process is done.

## Database

Normally, when starting a new node, you can just bootstrap from scratch. However, there are situations when you may prefer to reuse an existing database (ex: preserve keystore records, reduce sync time).

This tutorial will walk you through compressing your node's DB and moving it to another computer using `zip` and `scp`.

#### Database Backup

First, make sure to stop AvalancheGo, run:

```
sudo systemctl stop avalanchego
```

:::warning You must stop the Avalanche node before you back up the database otherwise data could become corrupted. :::

Once the node is stopped, you can `zip` the database directory to reduce the size of the backup and speed up the transfer using `scp`:

```
zip -r avalanche_db_backup.zip .avalanchego/db
```

*Note: It may take > 30 minutes to zip the node's DB.*

Next, you can transfer the backup to another machine:

```
scp -r ubuntu@PUBLICIP:/home/ubuntu/avalanche_db_backup.zip ~/avalanche_db_backup.zip
```

This assumes the username on the machine is `ubuntu`, replace with correct username in both places if it is different. Also, replace `PUBLICIP` with the actual public IP of the machine. If `scp` doesn't automatically use your downloaded SSH key, you can point to it manually:

```
scp -i /path/to/the/key.pem -r ubuntu@PUBLICIP:/home/ubuntu/avalanche_db_backup.zip ~/avalanche_db_backup.zip
```

Once executed, this command will create `avalanche_db_backup.zip` directory in your home directory.

#### Database Restore

*This tutorial assumes you have already completed "Database Backup" and have a backup at `~/avalanche_db_backup.zip`.*

First, we need to do the usual [installation](#) of the node. When the node is installed correctly, log into the machine where the node is running and stop it:

```
sudo systemctl stop avalanchego
```

:::warning You must stop the Avalanche node before you restore the database otherwise data could become corrupted. :::

We're ready to restore the database. First, let's move the DB on the existing node (you can remove this old DB later if the restore was successful):

```
mv .avalanchego/db .avalanchego/db-old
```

Next, we'll unzip the backup we moved from another node (this will place the unzipped files in `~/.avalanchego/db` when the command is run in the home directory):

```
unzip avalanche_db_backup.zip
```

After the database has been restored on a new node, use this command to start the node:

```
sudo systemctl start avalanchego
```

Node should now be running from the database on the new instance. To check that everything is in order and that node is not bootstrapping from scratch (which would indicate a problem), use:

```
sudo journalctl -u avalanchego -f
```

The node should be catching up to the network and fetching a small number of blocks before resuming normal operation (all the ones produced from the time when the node was stopped before the backup).

Once the backup has been restored and is working as expected, the zip can be deleted:

```
rm avalanche_db_backup.zip
```

### Database Direct Copy

You may be in a situation where you don't have enough disk space to create the archive containing the whole database, so you cannot complete the backup process as described previously.

In that case, you can still migrate your database to a new computer, by using a different approach: `direct copy`. Instead of creating the archive, moving the archive and unpacking it, we can do all of that on the fly.

To do so, you will need `ssh` access from the destination machine (where you want the database to end up) to the source machine (where the database currently is). Setting up `ssh` is the same as explained for `scp` earlier in the document.

Same as shown previously, you need to stop the node (on both machines):

```
sudo systemctl stop avalanchego
```

:::warning You must stop the Avalanche node before you back up the database otherwise data could become corrupted. :::

Then, on the destination machine, change to a directory where you would like to put the database files, enter the following command:

```
ssh -i /path/to/the/key.pem ubuntu@PUBLICIP 'tar czf - .avalanchego/db' | tar xvzf - -C .
```

Make sure to replace the correct path to the key, and correct IP of the source machine. This will compress the database, but instead of writing it to a file it will pipe it over `ssh` directly to destination machine, where it will be decompressed and written to disk. The process can take a long time, make sure it completes before continuing.

After copying is done, all you need to do now is move the database to the correct location on the destination machine. Assuming there is a default AvalancheGo node installation, we remove the old database and replace it with the new one:

```
rm -rf ~/.avalanchego/db
mv db ~/.avalanchego/db
```

You can now start the node on the destination machine:

```
sudo systemctl start avalanchego
```

Node should now be running from the copied database. To check that everything is in order and that node is not bootstrapping from scratch (which would indicate a problem), use:

```
sudo journalctl -u avalanchego -f
```

The node should be catching up to the network and fetching a small number of blocks before resuming normal operation (all the ones produced from the time when the node was stopped before the backup).

### Summary

Essential part of securing your node is the backup that enables full and painless restoration of your node. Following this tutorial you can rest easy knowing that should you ever find yourself in a situation where you need to restore your node from scratch, you can easily and quickly do so.

**If you have any problems following this tutorial, comments you want to share with us or just want to chat, you can reach us on our [Discord](#) server.**

**sidebar\_position: 8**

## Node Bootstrap

Node Bootstrap is the process where a node *securely* downloads linear chain blocks to recreate the latest state of the chain locally.

Bootstrap must guarantee that the local state of a node is in sync with the state of other valid nodes. Once bootstrap is completed, a node has the latest state of the chain and can verify new incoming transactions and reach consensus with other nodes, collectively moving forward the chains.

Bootstrapping a node is a multi-step process which requires downloading the chains required by the Primary Network (that is, the C-Chain, P-Chain, and X-Chain), as well as the chains required by any additional Subnets that the node explicitly tracks.

This document covers the high-level technical details of how bootstrapping works. This document glosses over some specifics, but the [AvalancheGo codebase](#) is open-source and is available for curious-minded readers to learn more.

### Validators and Where to Find Them

Bootstrapping is all about downloading all previously accepted containers *securely* so a node can have the latest correct state of the chain. A node can't arbitrarily trust any source - a malicious actor could provide malicious blocks, corrupting the bootstrapping node's local state, and making it impossible for the node to correctly validate the network and reach consensus with other correct nodes.

What's the most reliable source of information in the Avalanche ecosystem? It's a *large enough* majority of validators. Therefore, the first step of bootstrapping is finding a sufficient amount of validators to download containers from.

The P-Chain is responsible for all platform-level operations, including staking events that modify a Subnet's validator set. Whenever any chain (aside from the P-Chain itself) bootstraps, it requests an up-to-date validator set for that Subnet (Primary Network is a Subnet too). Once the Subnet's current validator set is known, the node can securely download containers from these validators to bootstrap the chain.

There is a caveat here: the validator set must be *up-to-date*. If a bootstrapping node's validator set is stale, the node may incorrectly believe that some nodes are still validators when their validation period has already expired. A node might unknowingly end up requesting blocks from non-validators which respond with malicious blocks that aren't safe to download.

**For this reason, every Avalanche node must fully bootstrap the P-chain first before moving on to the other Primary Network chains and other Subnets to guarantee that their validator sets are up-to-date.**

What about the P-chain? The P-chain can't ever have an up-to-date validator set before completing its bootstrap. To solve this chicken-and-egg situation the Avalanche Foundation maintains a trusted default set of validators called beacons (but users are free to configure their own). Beacon Node-IDs and IP addresses are listed in the [AvalancheGo codebase](#). Every node has the beacon list available from the start and can reach out to them as soon as it starts.

Validators are the only sources of truth for a blockchain. Validator availability is so key to the bootstrapping process that **bootstrapping is blocked until the node establishes a sufficient amount of secure connections to validators**. If the node fails to reach a sufficient amount within a given period of time, it shuts down as no operation can be carried out safely.

### Bootstrapping the Blockchain

Once a node is able to discover and connect to validator and beacon nodes, it's able to start bootstrapping the blockchain by downloading the individual containers.

One common misconception is that Avalanche blockchains are bootstrapped by retrieving containers starting at genesis and working up to the currently accepted frontier.

Instead, containers are downloaded from the accepted frontier downwards to genesis, and then their corresponding state transitions are executed upwards from genesis to the accepted frontier. The accepted frontier is the last accepted block for linear chains and the accepted vertices for DAGs.

Why can't nodes simply download blocks in chronological order, starting from genesis upwards? The reason is efficiency: if nodes downloaded containers upwards they would only get a safety guarantee by polling a majority of validators for every single container. That's a lot of network traffic for a single container, and a node would still need to do that for each container in the chain.

Instead, if a node starts by securely retrieving the accepted frontier from a majority of honest nodes and then recursively fetches the parent containers from the accepted frontier down to genesis, it can cheaply check that containers are correct just by verifying their IDs. Each Avalanche container has the IDs of its parents (one block parent for linear chains, possibly multiple parents for DAGs) and an ID's integrity can be guaranteed cryptographically.

Let's dive deeper into the two bootstrap phases - frontier retrieval and container execution.

#### Frontier Retrieval

The current frontier is retrieved by requesting them from validator or beacon nodes. Avalanche bootstrap is designed to be robust - it must be able to make progress even in the presence of slow validators or network failures. This process needs to be fault-tolerant to these types of failures, since bootstrapping may take quite some time to complete and network connections can be unreliable.

Bootstrap starts when a node has connected to a sufficient majority of validator stake. A node is able to start bootstrapping when it has connected to at least \$75% of total validator stake.

Seeders are the first set of peers that a node reaches out to when trying to figure out the current frontier. A subset of seeders is randomly sampled from the validator set. Seeders might be slow and provide a stale frontier, be malicious and return malicious container IDs, but they always provide an initial set of candidate frontiers to work with.

Once a node has received the candidate frontiers from its seeders, it polls **every network validator** to vet the candidates frontiers. It sends the list of candidate frontiers it received from the seeders to each validator, asking whether or not they know about these frontiers. Each validator responds returning the subset of known candidates, regardless of how up-to-date or stale the containers are. Each validator returns containers irrespective of their age so that bootstrap works even in the presence of a stale frontier.

Frontier retrieval is completed when at least one of the candidate frontiers is supported by at least \$50% of total validator stake. Multiple candidate frontiers may be supported by a majority of stake, after which point the next phase, container fetching starts.

At any point in these steps a network issue may occur, preventing a node from retrieving or validating frontiers. If this occurs, bootstrap restarts by sampling a new set of seeders and repeating the bootstrapping process, optimistically assuming that the network issue will go away.

## Containers Execution

Once a node has at least one valid frontiers, it starts downloading parent containers for each frontier. If it's the first time the node is running, it won't know about any containers and will try fetching all parent containers recursively from the accepted frontier down to genesis (unless [state sync](#) is enabled). If bootstrap had already run previously, some containers are already available locally and the node will stop as soon as it finds a known one.

A node first just fetches and parses containers. Once the chain is complete, the node executes them in chronological order starting from the earliest downloaded container to the accepted frontier. This allows the node to rebuild the full chain state and to eventually be in sync with the rest of the network.

## When Does Bootstrapping Finish?

You've seen how [bootstrap works](#) for a single chain. However, a node must bootstrap the chains in the Primary Network as well as the chains in each Subnet it tracks. This begs the questions - when are these chains bootstrapped? When is a node done bootstrapping?

The P-chain is always the first to bootstrap before any other chain. Once the P-Chain has finished, all other chains start bootstrapping in parallel, connecting to their own validators independently of one another.

A node completes bootstrapping a Subnet once all of its corresponding chains have completed bootstrapping. Because the Primary Network is a special case of Subnet that includes the entire network, this applies to it as well as any other manually tracked Subnets.

Note that Subnets bootstrap is independently of one another - so even if one Subnet has bootstrapped and is validating new transactions and adding new containers, other Subnets may still be bootstrapping in parallel.

Within a single Subnet however, a Subnet isn't done bootstrapping until the last chain completes bootstrapping. It's possible for a single chain to effectively stall a node from finishing the bootstrap for a single Subnet, if it has a sufficiently long history or each operation is complex and time consuming. Even worse, other Subnet validators are continuously accepting new transactions and adding new containers on top of the previously known frontier, so a node that's slow to bootstrap can continuously fall behind the rest of the network.

Nodes mitigate this by restarting bootstrap for any chains which is blocked waiting for the remaining Subnet chains to finish bootstrapping. These chains repeat the frontier retrieval and container downloading phases to stay up-to-date with the Subnet's ever moving current frontier until the slowest chain has completed bootstrapping.

Once this is complete, a node is finally ready to validate the network.

## State Sync

The full node bootstrap process is long, and gets longer and longer over time as more and more containers are accepted. Nodes need to bootstrap a chain by reconstructing the full chain state locally - but downloading and executing each container isn't the only way to do this.

Starting from [AvalancheGo version 1.7.11](#), nodes can use state sync to drastically cut down bootstrapping time on the C-Chain. Instead of executing each block, state sync uses cryptographic techniques to download and verify just the state associated with the current frontier. State synced nodes can't serve every C-chain block ever historically accepted, but they can safely retrieve the full C-chain state needed to validate in a much shorter time. State sync will fetch the previous 256 blocks prior to support the previous block hash operation code.

State sync is currently only available for the C-chain. The P-chain and X-chain currently bootstrap by downloading all blocks. Note that irrespective of the bootstrap method used (including state sync), each chain is still blocked on all other chains in its Subnet completing their bootstrap before continuing into normal operation.

:::note

There are no configs to state sync an archival node. If you need all the historical state then you must not use state sync and setup the config of the node for an archival node.

:::

## Conclusions and FAQ

If you got this far, you've hopefully gotten a better idea of what's going on when your node bootstraps. Here's a few frequently asked questions about bootstrapping.

### How Can I Get the ETA for Node Bootstrap?

Logs provide information about both container downloading and their execution for each chain. Here is an example

```
[02-16|17:31:42.950] INFO <P Chain> bootstrap/bootstrap.go:494 fetching blocks {"numFetchedBlocks": 5000, "numTotalBlocks": 101357, "eta": "2m52s"}
[02-16|17:31:58.110] INFO <P Chain> bootstrap/bootstrap.go:494 fetching blocks {"numFetchedBlocks": 10000, "numTotalBlocks": 101357, "eta": "3m40s"}
[02-16|17:32:04.554] INFO <P Chain> bootstrap/bootstrap.go:494 fetching blocks {"numFetchedBlocks": 15000, "numTotalBlocks": 101357, "eta": "2m56s"}
...
[02-16|17:36:52.404] INFO <P Chain> queue/jobs.go:203 executing operations {"numExecuted": 17881, "numToExecute": 101357, "eta": "2m20s"}
[02-16|17:37:22.467] INFO <P Chain> queue/jobs.go:203 executing operations {"numExecuted": 35009, "numToExecute": 101357, "eta": "1m54s"}
[02-16|17:37:52.468] INFO <P Chain> queue/jobs.go:203 executing operations {"numExecuted": 52713, "numToExecute": 101357, "eta": "1m23s"}
```

Similar logs are emitted for X and C chains and any chain in explicitly tracked Subnets.

### Why Chain Bootstrap ETA Keeps On Changing?

As you saw in the [bootstrap completion section](#), a Subnet like the Primary Network completes once all of its chains finish bootstrapping. Some Subnet chains may have to wait for the slowest to finish. They'll restart bootstrapping in the meantime, to make sure they won't fall back too much with respect to the network accepted frontier.

### Why Are AvalancheGo APIs Disabled During Bootstrapping?

AvalancheGo APIs are [explicitly disabled](#) during bootstrapping. The reason is that if the node has not fully rebuilt its Subnets state, it can't provide accurate information. AvalancheGo APIs are activated once bootstrap completes and node transition into its normal operating mode, accepting and validating transactions.

**sidebar\_position: 7 description: In this doc, learn how to run offline pruning on your node to reduce its disk usage.**

## Run C-Chain Offline Pruning

### Introduction

Offline Pruning is ported from `go-ethereum` to reduce the amount of disk space taken up by the TrieDB (storage for the Merkle Forest).

Offline pruning creates a bloom filter and adds all trie nodes in the active state to the bloom filter to mark the data as protected. This ensures that any part of the active state will not be removed during offline pruning.

After generating the bloom filter, offline pruning iterates over the database and searches for trie nodes that are safe to be removed from disk.

A bloom filter is a probabilistic data structure that reports whether an item is definitely not in a set or possibly in a set. Therefore, for each key we iterate, we check if it is in the bloom filter. If the key is definitely not in the bloom filter, then it is not in the active state and we can safely delete it. If the key is possibly in the set, then we skip over it to ensure we do not delete any active state.

During iteration, the underlying database (LevelDB) writes deletion markers, causing a temporary increase in disk usage.

After iterating over the database and deleting any old trie nodes that it can, offline pruning then runs compaction to minimize the DB size after the potentially large number of delete operations.

### Finding the C-Chain Config File

In order to enable offline pruning, you need to update the C-Chain config file to include the parameters `offline-pruning-enabled` and `offline-pruning-data-directory`.

The default location of the C-Chain config file is `~/.avalanchego/configs/chains/C/config.json`. Please note that by default, this file does not exist. You would need to create it manually. You can update the directory for chain configs by passing in the directory of your choice via the CLI argument: `chain-config-dir`. See [this](#) for more info. For example, if you start your node with:

```
./build/avalanchego --chain-config-dir=/home/ubuntu/chain-configs
```

The chain config directory will be updated to `/home/ubuntu/chain-configs` and the corresponding C-Chain config file will be: `/home/ubuntu/chain-configs/C/config.json`.

### Running Offline Pruning

In order to enable offline pruning, update the C-Chain config file to include the following parameters:

```
{
 "offline-pruning-enabled": true,
 "offline-pruning-data-directory": "/home/ubuntu/offline-pruning"
}
```

This will set `/home/ubuntu/offline-pruning` as the directory to be used by the offline pruner. Offline pruning will store the bloom filter in this location, so you must ensure that the path exists.

Now that the C-Chain config file has been updated, you can start your node with the command (no CLI arguments are necessary if using the default chain config directory):

```
./build/avalanchego
```

Once AvalancheGo starts the C-Chain, you can expect to see update logs from the offline pruner:

```
INFO [02-09|00:20:15.625] Iterating state snapshot
INFO [02-09|00:20:23.626] Iterating state snapshot
accounts=297,231 slots=6,669,708 elapsed=16.001s eta=1m29.03s
accounts=401,907 slots=10,698,094 elapsed=24.001s
eta=1m32.522s
```

```

INFO [02-09|00:20:31.626] Iterating state snapshot
eta=1m10.927s
INFO [02-09|00:20:39.626] Iterating state snapshot
eta=1m2.603s
INFO [02-09|00:20:47.626] Iterating state snapshot
eta=1m8.834s
INFO [02-09|00:20:55.626] Iterating state snapshot
eta=57.401s
INFO [02-09|00:21:03.626] Iterating state snapshot
eta=47.674s
INFO [02-09|00:21:11.626] Iterating state snapshot
eta=45.185s
INFO [02-09|00:21:19.626] Iterating state snapshot
eta=34.59s
INFO [02-09|00:21:27.627] Iterating state snapshot
eta=25.006s
INFO [02-09|00:21:35.627] Iterating state snapshot
eta=20.052s
INFO [02-09|00:21:43.627] Iterating state snapshot
eta=9.299s
INFO [02-09|00:21:47.342] Iterated snapshot
INFO [02-09|00:21:47.351] Writing state bloom to disk
pruning/statebloom.0xd6fca36db4b60b34330377040ef6566f6033ed8464731cbb06dc35c8401fa38e.bf.gz
INFO [02-09|00:23:04.421] State bloom filter committed
pruning/statebloom.0xd6fca36db4b60b34330377040ef6566f6033ed8464731cbb06dc35c8401fa38e.bf.gz

```

The bloom filter should be populated and committed to disk after about 5 minutes. At this point, if the node shuts down, it will resume the offline pruning session when it restarts (note: this operation cannot be cancelled).

In order to ensure that users do not mistakenly leave offline pruning enabled for the long term (which could result in an hour of downtime on each restart), we have added a manual protection which requires that after an offline pruning session, the node must be started with offline pruning disabled at least once before it will start with offline pruning enabled again. Therefore, once the bloom filter has been committed to disk, you should update the C-Chain config file to include the following parameters:

```
{
 "offline-pruning-enabled": false,
 "offline-pruning-data-directory": "/home/ubuntu/offline-pruning"
}
```

It is important to keep the same data directory in the config file, so that the node knows where to look for the bloom filter on a restart if offline pruning has not finished.

Now if your node restarts, it will be marked as having correctly disabled offline pruning after the run and be allowed to resume normal operation once offline pruning has finished running.

You will see progress logs throughout the offline pruning run which will indicate the session's progress:

```

INFO [02-09|00:31:51.920] Pruning state data
eta=12m50.961s
INFO [02-09|00:31:59.921] Pruning state data
eta=12m13.822s
INFO [02-09|00:32:07.921] Pruning state data
eta=12m23.915s
INFO [02-09|00:32:15.921] Pruning state data
eta=12m33.965s
INFO [02-09|00:32:23.921] Pruning state data
eta=12m44.004s
INFO [02-09|00:32:31.921] Pruning state data
eta=12m54.01s
INFO [02-09|00:32:39.921] Pruning state data
eta=13m3.992s
INFO [02-09|00:32:47.922] Pruning state data
eta=13m13.951s
INFO [02-09|00:32:55.922] Pruning state data
eta=13m23.885s
INFO [02-09|00:33:03.923] Pruning state data
eta=13m33.79s
INFO [02-09|00:33:11.923] Pruning state data
eta=13m43.678s
INFO [02-09|00:33:19.924] Pruning state data
eta=13m53.551s
INFO [02-09|00:33:27.924] Pruning state data
eta=14m3.389s
INFO [02-09|00:33:35.924] Pruning state data
eta=14m13.192s
INFO [02-09|00:33:43.925] Pruning state data
eta=14m22.976s
INFO [02-09|00:33:51.925] Pruning state data
nodes=40,116,759 size=10.08GiB elapsed=8m47.499s
nodes=41,659,059 size=10.47GiB elapsed=8m55.499s
nodes=41,687,047 size=10.48GiB elapsed=9m3.499s
nodes=41,715,823 size=10.48GiB elapsed=9m11.499s
nodes=41,744,167 size=10.49GiB elapsed=9m19.500s
nodes=41,772,613 size=10.50GiB elapsed=9m27.500s
nodes=41,801,267 size=10.50GiB elapsed=9m35.500s
nodes=41,829,714 size=10.51GiB elapsed=9m43.500s
nodes=41,858,400 size=10.52GiB elapsed=9m51.501s
nodes=41,887,131 size=10.53GiB elapsed=9m59.501s
nodes=41,915,583 size=10.53GiB elapsed=10m7.502s
nodes=41,943,891 size=10.54GiB elapsed=10m15.502s
nodes=41,972,281 size=10.55GiB elapsed=10m23.502s
nodes=42,001,414 size=10.55GiB elapsed=10m31.503s
nodes=42,029,987 size=10.56GiB elapsed=10m39.504s
nodes=42,777,042 size=10.75GiB elapsed=10m47.504s

```

```

eta=14m7.245s
INFO [02-09|00:34:00.950] Pruning state data nodes=42,865,413 size=10.77GiB elapsed=10m56.529s
eta=14m15.927s
INFO [02-09|00:34:08.956] Pruning state data nodes=42,918,719 size=10.79GiB elapsed=11m4.534s
eta=14m24.453s
INFO [02-09|00:34:22.816] Pruning state data nodes=42,952,925 size=10.79GiB elapsed=11m18.394s
eta=14m41.243s
INFO [02-09|00:34:30.818] Pruning state data nodes=42,998,715 size=10.81GiB elapsed=11m26.397s
eta=14m49.961s
INFO [02-09|00:34:38.828] Pruning state data nodes=43,046,476 size=10.82GiB elapsed=11m34.407s
eta=14m58.572s
INFO [02-09|00:34:46.893] Pruning state data nodes=43,107,656 size=10.83GiB elapsed=11m42.472s
eta=15m6.729s
INFO [02-09|00:34:55.038] Pruning state data nodes=43,168,834 size=10.85GiB elapsed=11m50.616s
eta=15m14.934s
INFO [02-09|00:35:03.039] Pruning state data nodes=43,446,900 size=10.92GiB elapsed=11m58.618s
eta=15m14.705s

```

When the node completes, it will emit the following log and resume normal operation:

```

INFO [02-09|00:42:16.009] Pruning state data nodes=93,649,812 size=23.53GiB elapsed=19m11.588s
eta=1m2.658s
INFO [02-09|00:42:24.009] Pruning state data nodes=95,045,956 size=23.89GiB elapsed=19m19.588s
eta=45.149s
INFO [02-09|00:42:32.009] Pruning state data nodes=96,429,410 size=24.23GiB elapsed=19m27.588s
eta=28.041s
INFO [02-09|00:42:40.009] Pruning state data nodes=97,811,804 size=24.58GiB elapsed=19m35.588s
eta=11.204s
INFO [02-09|00:42:45.359] Pruned state data nodes=98,744,430 size=24.82GiB elapsed=19m40.938s
INFO [02-09|00:42:45.360] Compacting database range=0x00-0x10 elapsed="2.157μs"
INFO [02-09|00:43:12.311] Compacting database range=0x10-0x20 elapsed=26.951s
INFO [02-09|00:43:38.763] Compacting database range=0x20-0x30 elapsed=53.402s
INFO [02-09|00:44:04.847] Compacting database range=0x30-0x40 elapsed=1m19.496s
INFO [02-09|00:44:31.194] Compacting database range=0x40-0x50 elapsed=1m45.834s
INFO [02-09|00:45:31.580] Compacting database range=0x50-0x60 elapsed=2m46.220s
INFO [02-09|00:45:58.465] Compacting database range=0x60-0x70 elapsed=3m13.104s
INFO [02-09|00:51:17.593] Compacting database range=0x70-0x80 elapsed=8m32.233s
INFO [02-09|00:56:19.679] Compacting database range=0x80-0x90 elapsed=13m34.319s
INFO [02-09|00:56:46.011] Compacting database range=0x90-0xa0 elapsed=14m0.651s
INFO [02-09|00:57:12.370] Compacting database range=0xa0-0xb0 elapsed=14m27.010s
INFO [02-09|00:57:38.600] Compacting database range=0xb0-0xc0 elapsed=14m53.239s
INFO [02-09|00:58:06.311] Compacting database range=0xc0-0xd0 elapsed=15m20.951s
INFO [02-09|00:58:35.484] Compacting database range=0xd0-0xe0 elapsed=15m50.123s
INFO [02-09|00:59:05.449] Compacting database range=0xe0-0xf0 elapsed=16m20.089s
INFO [02-09|00:59:34.365] Compacting database range=0xf0- elapsed=16m49.005s
INFO [02-09|00:59:34.367] Database compaction finished elapsed=16m49.006s
INFO [02-09|00:59:34.367] State pruning successful pruned=24.82GiB elapsed=39m34.749s
INFO [02-09|00:59:34.367] Completed offline pruning. Re-initializing blockchain.
INFO [02-09|00:59:34.387] Loaded most recent local header number=10,671,401 hash=b52d0a..7bd166 age=40m29s
INFO [02-09|00:59:34.387] Loaded most recent local full block number=10,671,401 hash=b52d0a..7bd166 age=40m29s
INFO [02-09|00:59:34.387] Initializing snapshots async=true
DEBUG[02-09|00:59:34.390] Reinjecting stale transactions count=0
INFO [02-09|00:59:34.395] Transaction pool price threshold updated price=470,000,000,000
INFO [02-09|00:59:34.396] Transaction pool price threshold updated price=225,000,000,000
INFO [02-09|00:59:34.396] Transaction pool price threshold updated price=0
INFO [02-09|00:59:34.396] lastAccepted = 0xb52d0a1302e405b5487c3a0243106b5e13a915c6e178da9f8491cebf017bd166
INFO [02-09|00:59:34] <C Chain> snow/engine/snowman/transitive.go#67: initializing consensus engine
INFO [02-09|00:59:34] <C Chain> snow/engine/snowman/bootstrap/bootstrap.go#220: Starting bootstrap...
INFO [02-09|00:59:34] chains/manager.go#246: creating chain:
 ID: 2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM
 VMID:jvvyf0TxGMDLJLGW55kdP2pZzSUYsQ5Raupu4TW342AUABtq
INFO [02-09|00:59:34.425] Enabled APIs: eth, eth-filter, net, web3, internal-eth, internal-blockchain, internal-transaction, avax
DEBUG[02-09|00:59:34.425] Allowed origin(s) for WS RPC interface [*]
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2q9e4r6Mu3U68nUi1fYjgbR6JvwrRx36CohpAX5Uqxse55x1Q5/avax
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2q9e4r6Mu3U68nUi1fYjgbR6JvwrRx36CohpAX5Uqxse55x1Q5/rpc
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2q9e4r6Mu3U68nUi1fYjgbR6JvwrRx36CohpAX5Uqxse55x1Q5/ws
INFO [02-09|00:59:34] <X Chain> vms/avm/vm.go#437: Fee payments are using Asset with Alias: AVAX, AssetID: FvwEAhmxfieiG88snVgq2hC6whRyY3EFYAvembMgDNDCGxN5Z
INFO [02-09|00:59:34] <X Chain> vms/avm/vm.go#229: address transaction indexing is disabled
INFO [02-09|00:59:34] <X Chain> snow/engine/avalanche/transitive.go#71: initializing consensus engine
INFO [02-09|00:59:34] <X Chain> snow/engine/avalanche/bootstrap/bootstrap.go#258: Starting bootstrap...
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM
INFO [02-09|00:59:34] <P Chain> snow/engine/snowman/bootstrap/bootstrap.go#445: waiting for the remaining chains in this subnet to finish syncing
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM/wallet
INFO [02-09|00:59:34] api/server/server.go#203: adding route /ext/bc/2oYMBNV4eNHyqk2fjjV5nVQLDbtmNJzq5s3qs3Lo6ftnC6FBByM/events

```

```

INFO [02-09|00:59:34] <P Chain> snow/engine/common/bootstrapper.go#235: Bootstrapping started syncing with 1 vertices in the accepted frontier
INFO [02-09|00:59:46] <X Chain> snow/engine/common/bootstrapper.go#235: Bootstrapping started syncing with 2 vertices in the accepted frontier
INFO [02-09|00:59:49] <C Chain> snow/engine/common/bootstrapper.go#235: Bootstrapping started syncing with 1 vertices in the accepted frontier
INFO [02-09|00:59:49] <X Chain> snow/engine/avalanche/bootstrap/bootstrapper.go#473: bootstrapping fetched 55 vertices. Executing transaction state transitions...
INFO [02-09|00:59:49] <X Chain> snow/engine/common/queue/jobs.go#171: executed 55 operations
INFO [02-09|00:59:49] <X Chain> snow/engine/avalanche/bootstrap/bootstrapper.go#484: executing vertex state transitions...
INFO [02-09|00:59:49] <X Chain> snow/engine/common/queue/jobs.go#171: executed 55 operations
INFO [02-09|01:00:07] <C Chain> snow/engine/snowman/bootstrap/bootstrapper.go#406: bootstrapping fetched 1241 blocks. Executing state transitions...

```

At this point, the node will go into bootstrapping and (once bootstrapping completes) resume consensus and operate as normal.

## Disk Space Considerations

To ensure the node does not enter an inconsistent state, the bloom filter used for pruning is persisted to `offline-pruning-data-directory` for the duration of the operation. This directory should have `offline-pruning-bloom-filter-size` available in disk space (default 512 MB).

The underlying database (LevelDB) uses deletion markers (tombstones) to identify newly deleted keys. These markers are temporarily persisted to disk until they are removed during a process known as compaction. This will lead to an increase in disk usage during pruning. If your node runs out of disk space during pruning, you may safely restart the pruning operation. This may succeed as restarting the node triggers compaction.

**If restarting the pruning operation does not succeed, additional disk space should be provisioned.**

**sidebar\_position: 3**

## Monitor an Avalanche Node

### Introduction

This tutorial will show how to set up infrastructure to monitor an instance of [AvalancheGo](#). We will use:

- [Prometheus](#) to gather and store data
- [node\\_exporter](#) to get information about the machine,
- AvalancheGo's [metrics API](#) to get information about the node
- [Grafana](#) to visualize data on a dashboard.
- A set of pre-made [Avalanche dashboards](#)

Prerequisites:

- A running AvalancheGo node
- Shell access to the machine running the node
- Administrator privileges on the machine

This tutorial assumes you have Ubuntu 20.04 running on your node. Other Linux flavors that use `systemd` for running services and `apt-get` for package management might work but have not been tested. Community member has reported it works on Debian 10, might work on other Debian releases as well.

### Caveat: Security

:::danger

The system as described here **should not** be opened to the public internet. Neither Prometheus nor Grafana as shown here is hardened against unauthorized access. Make sure that both of them are accessible only over a secured proxy, local network, or VPN. Setting that up is beyond the scope of this tutorial, but exercise caution. Bad security practices could lead to attackers gaining control over your node! It is your responsibility to follow proper security practices.

:::

### Monitoring Installer Script

In order to make node monitoring easier to install, we have made a script that does most of the work for you. To download and run the script, log into the machine the node runs on with a user that has administrator privileges and enter the following command:

```

wget -nd -m https://raw.githubusercontent.com/ava-labs/avalanche-monitoring/main/grafana/monitoring-installer.sh ;\
chmod 755 monitoring-installer.sh;

```

This will download the script and make it executable.

Script itself is run multiple times with different arguments, each installing a different tool or part of the environment. To make sure it downloaded and set up correctly, begin by running:

```

./monitoring-installer.sh --help

```

It should display:

```

Usage: ./monitoring-installer.sh [--1|--2|--3|--4|--5|--help]

Options:
--help Shows this message
--1 Step 1: Installs Prometheus
--2 Step 2: Installs Grafana
--3 Step 3: Installs node_exporter
--4 Step 4: Installs AvalancheGo Grafana dashboards
--5 Step 5: (Optional) Installs additional dashboards

Run without any options, script will download and install latest version of AvalancheGo dashboards.

```

Let's get to it.

## Step 1: Set up Prometheus

Run the script to execute the first step:

```
./monitoring-installer.sh --1
```

It should produce output something like this:

```

AvalancheGo monitoring installer

STEP 1: Installing Prometheus

Checking environment...
Found arm64 architecture...
Prometheus install archive found:
https://github.com/prometheus/prometheus/releases/download/v2.31.0/prometheus-2.31.0.linux-arm64.tar.gz
Attempting to download:
https://github.com/prometheus/prometheus/releases/download/v2.31.0/prometheus-2.31.0.linux-arm64.tar.gz
prometheus.tar.gz 100%
[=====] 65.11M 123MB/s in 0.5s
2021-11-05 14:16:11 URL:https://github-releases.githubusercontent.com/6838921/a215b0e7-df1f-402b-9541-a3ec9d431f76?X-Amz-
Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211105%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
Date=20211105T141610Z&X-Amz-Expires=300&X-Amz-Signature=72a8ae4c6b5cea962bb9cad242cb4478082594b484d6a519de58b8241b319d94&X-Amz-
SignedHeaders=host&actor_id=0&key_id=0&repo_id=6838921&response-content-disposition=attachment%3B%20filename%3Dprometheus-
2.31.0.linux-arm64.tar.gz&response-content-type=application%2Foctet-stream [68274531/68274531] -> "prometheus.tar.gz" [1]
...

```

You may be prompted to confirm additional package installs, do that if asked. Script run should end with instructions on how to check that Prometheus installed correctly. Let's do that, run:

```
sudo systemctl status prometheus
```

It should output something like:

```

● prometheus.service - Prometheus
 Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset: enabled)
 Active: active (running) since Fri 2021-11-12 11:38:32 UTC; 17min ago
 Docs: https://prometheus.io/docs/introduction/overview/
 Main PID: 548 (prometheus)
 Tasks: 10 (limit: 9300)
 Memory: 95.6M
 CGroup: /system.slice/prometheus.service
 └─548 /usr/local/bin/prometheus --config.file=/etc/prometheus/prometheus.yml --storage.tsdb.path=/var/lib/prometheus --
 web.console.templates=/etc/prometheus/con>

Nov 12 11:38:33 ip-172-31-36-200 prometheus[548]: ts=2021-11-12T11:38:33.644Z caller=head.go:590 level=info component=tsdb
msg="WAL segment loaded" segment=81 maxSegment=84
Nov 12 11:38:33 ip-172-31-36-200 prometheus[548]: ts=2021-11-12T11:38:33.773Z caller=head.go:590 level=info component=tsdb
msg="WAL segment loaded" segment=82 maxSegment=84

```

Note the `active (running)` status (press `q` to exit). You can also check Prometheus web interface, available on <http://your-node-host-ip:9090/>

:::warning

You may need to do `sudo ufw allow 9090/tcp` if the firewall is on, and/or adjust the security settings to allow connections to port 9090 if the node is running on a cloud instance. For AWS, you can look it up [here](#). If on public internet, make sure to only allow your IP to connect!

:::

If everything is OK, let's move on.

## Step 2: Install Grafana

Run the script to execute the second step:

```
./monitoring-installer.sh --2
```

It should produce output something like this:

```
AvalancheGo monitoring installer

STEP 2: Installing Grafana

OK
deb https://packages.grafana.com/oss/deb stable main
Hit:1 http://us-east-2.ec2.ports.ubuntu.com/ubuntu-ports focal InRelease
Get:2 http://us-east-2.ec2.ports.ubuntu.com/ubuntu-ports focal-updates InRelease [114 kB]
Get:3 http://us-east-2.ec2.ports.ubuntu.com/ubuntu-ports focal-backports InRelease [101 kB]
Hit:4 http://ppa.launchpad.net/longsleep/golang-backports/ubuntu focal InRelease
Get:5 http://ports.ubuntu.com/ubuntu-ports focal-security InRelease [114 kB]
Get:6 https://packages.grafana.com/oss/deb stable InRelease [12.1 kB]
...
...
```

To make sure it's running properly:

```
sudo systemctl status grafana-server
```

which should again show Grafana as `active`. Grafana should now be available at `http://your-node-host-ip:3000/` from your browser. Log in with username: admin, password: admin, and you will be prompted to set up a new, secure password. Do that.

:::warning

You may need to do `sudo ufw allow 3000/tcp` if the firewall is on, and/or adjust the cloud instance settings to allow connections to port 3000. If on public internet, make sure to only allow your IP to connect!

:::

Prometheus and Grafana are now installed, we're ready for the next step.

## Step 3: Set up node\_exporter

In addition to metrics from AvalancheGo, let's set up monitoring of the machine itself, so we can check CPU, memory, network and disk usage and be aware of any anomalies. For that, we will use `node_exporter`, a Prometheus plugin.

Run the script to execute the third step:

```
./monitoring-installer.sh --3
```

The output should look something like this:

```
AvalancheGo monitoring installer

STEP 3: Installing node_exporter

Checking environment...
Found arm64 architecture...
Downloading archive...
https://github.com/prometheus/node_exporter/releases/download/v1.2.2/node_exporter-1.2.2.linux-arm64.tar.gz
node_exporter.tar.gz
[=====] 100% 7.91M --.-KB/s in 0.1s
2021-11-05 14:57:25 URL:https://github-releases.githubusercontent.com/9524057/6dc22304-a1f5-419b-b296-906f6ddc168dc?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20211105%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20211105T145725Z&X-Amz-Expires=300&X-Amz-Signature=3890e09e58ea9d4180684d9286c9e791b96b0c411d8f8a494f77e99f260bdccb4X-Amz-SignedHeaders=host&actor_id=0&key_id=0&repo_id=9524057&response-content-disposition=attachment%3B%20filename%3Dnode_exporter-1.2.2.linux-arm64.tar.gz&response-content-type=application%2Foctet-stream [8296266/8296266] -> "node_exporter.tar.gz" [1]
node_exporter-1.2.2.linux-arm64/LICENSE
```

Again, we check that the service is running correctly:

```
sudo systemctl status node_exporter
```

If the service is running, Prometheus, Grafana and `node_exporter` should all work together now. To check, in your browser visit Prometheus web interface on `http://your-node-host-ip:9090/targets`. You should see three targets enabled:

- Prometheus
- AvalancheGo

```
• avalanchego-machine
```

Make sure that all of them have `State` as `UP`.

:::info

If you run your AvalancheGo node with TLS enabled on your API port, you will need to manually edit the `/etc/prometheus/prometheus.yml` file and change the `avalanchego` job to look like this:

```
- job_name: "avalanchego"
 metrics_path: "/ext/metrics"
 scheme: "https"
 tls_config:
 insecure_skip_verify: true
 static_configs:
 - targets: ["localhost:9650"]
```

Mind the spacing (leading spaces too)! You will need admin privileges to do that (use `sudo`). Restart Prometheus service afterwards with `sudo systemctl restart prometheus`.

:::

All that's left to do now is to provision the data source and install the actual dashboards that will show us the data.

## Step 4: Dashboards

Run the script to install the dashboards:

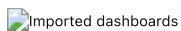
```
./monitoring-installer.sh --4
```

It will produce output something like this:

```
AvalancheGo monitoring installer

Download...
Last-modified header missing -- time-stamps turned off.
2021-11-05 14:57:47 URL:https://raw.githubusercontent.com/ava-labs/avalanche-monitoring/master/grafana/dashboards/c_chain.json
[50282/50282] -> "c_chain.json" [1]
FINISHED --2021-11-05 14:57:47--
Total wall clock time: 0.2s
Downloaded: 1 files, 49K in 0s (132 MB/s)
Last-modified header missing -- time-stamps turned off.
...
```

This will download the latest versions of the dashboards from GitHub and provision Grafana to load them, as well as defining Prometheus as a data source. It may take up to 30 seconds for the dashboards to show up. In your browser, go to: `http://your-node-host-ip:3000/dashboards`. You should see 7 Avalanche dashboards:



Select 'Avalanche Main Dashboard' by clicking its title. It should load, and look similar to this:



Some graphs may take some time to populate fully, as they need a series of data points in order to render correctly.

You can bookmark the main dashboard as it shows the most important information about the node at a glance. Every dashboard has a link to all the others as the first row, so you can move between them easily.

## Step 5: Additional Dashboards (Optional)

Step 4 installs the basic set of dashboards that make sense to have on any node. Step 5 is for installing additional dashboards that may not be useful for every installation.

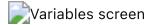
Currently, there is only one additional dashboard: Subnets. If your node is running any Subnets, you may want to add this as well. Do:

```
./monitoring-installer.sh --5
```

This will add the Subnets dashboard. It allows you to monitor operational data for any Subnet that is synced on the node. There is a Subnet switcher that allows you to switch between different Subnets. As there are many Subnets and not every node will have all of them, by default, it comes populated only with Spaces and WAGMI Subnets that exist on Fuji testnet:



To configure the dashboard and add any Subnets that your node is syncing, you will need to edit the dashboard. Select the `dashboard settings` icon (image of a cog) in the upper right corner of the dashboard display and switch to `Variables` section and select the `subnet` variable. It should look something like this:



The variable format is:

```
Subnet name:<BlockchainID>
```

and the separator between entries is a comma. Entries for Spaces and WAGMI look like:

```
Spaces (Fuji) : 2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeagh6tt, WAGMI (Fuji) :
2AM3vsuLoJdGBGqX2ibE8RGEq4Lg7g4bot6BT1Z7B9dH5corUD
```

After editing the values, press `Update` and then click `Save dashboard` button and confirm. Press the back arrow in the upper left corner to return to the dashboard. New values should now be selectable from the dropdown and data for the selected Subnet will be shown in the panels.

## Updating

Available node metrics are updated constantly, new ones are added and obsolete removed, so it is good a practice to update the dashboards from time to time, especially if you notice any missing data in panels. Updating the dashboards is easy, just run the script with no arguments, and it will refresh the dashboards with the latest available versions. Allow up to 30s for dashboards to update in Grafana.

If you added the optional extra dashboards (step 5), they will be updated as well.

## Summary

Using the script to install node monitoring is easy, and it gives you insight into how your node is behaving and what's going on under the hood. Also, pretty graphs!

**If you have feedback on this tutorial, problems with the script or following the steps, send us a message on [Discord](#).**

**sidebar\_position: 6 description: In this doc, learn how to run offline pruning on your node to reduce its disk usage.**

## Subnet Configs

It is possible to provide parameters for a Subnet. Parameters here apply to all chains in the specified Subnet.

AvalancheGo looks for files specified with `{subnetID}.json` under `--subnet-config-dir` as documented [here](#).

Here is an example of Subnet config file:

```
{
 "validatorOnly": false,
 "consensusParameters": {
 "k": 25,
 "alpha": 18
 },
 "appGossipNonValidatorSize": 10
}
```

## Parameters

### Private Subnet

`validatorOnly (bool)`

If `true` this node does not expose Subnet blockchain contents to non Validators via P2P messages. Defaults to `false`.

Avalanche Subnets are public by default. It means that every node can sync and listen ongoing transactions/blocks in Subnets, even they're not validating the listened Subnet.

Subnet validators can choose not to publish contents of blockchains via this configuration. If a node sets `validatorOnly` to true, the node exchanges messages only with this Subnet's validators. Other peers will not be able to learn contents of this Subnet from this node.

:::tip

This is a node-specific configuration. Every validator of this Subnet has to use this configuration in order to create a full private Subnet.

...

`allowedNodes (string list)`

If `validatorOnly=true` this allows explicitly specified NodeIDs to be allowed to sync the Subnet regardless of validator status. Defaults to be empty.

:::tip

This is a node-specific configuration. Every validator of this Subnet has to use this configuration in order to properly allow a node in the private Subnet.

:::

#### **proposerMinBlockDelay (duration)**

The minimum delay performed when building snowman++ blocks. Default is set to 1 second.

As one of the ways to control network congestion, Snowman++ will only build a block `proposerMinBlockDelay` after the parent block's timestamp. Some high-performance custom VM may find this too strict. This flag allows tuning the frequency at which blocks are built.

### **Consensus Parameters**

Subnet configs supports loading new consensus parameters. JSON keys are different from their matching `CLI` keys. These parameters must be grouped under `consensusParameters` key. The consensus parameters of a Subnet default to the same values used for the Primary Network, which are given [CLI Show Parameters](#).

| CLI Key                          | JSON Key              |
|----------------------------------|-----------------------|
| --snow-sample-size               | k                     |
| --snow-quorum-size               | alpha                 |
| --snow-virtuous-commit-threshold | betaVirtuous          |
| --snow-rogue-commit-threshold    | betaRogue             |
| --snow-concurrent-repolls        | concurrentRepolls     |
| --snow-optimal-processing        | optimalProcessing     |
| --snow-max-processing            | maxOutstandingItems   |
| --snow-max-time-processing       | maxItemProcessingTime |
| --snow-avalanche-batch-size      | batchSize             |
| --snow-avalanche-num-parents     | parentSize            |

### **Gossip Configs**

It's possible to define different Gossip configurations for each Subnet without changing values for Primary Network. For example in Primary Network transaction mempools are not gossiped to non Validators ( `--consensus-app-gossip-non-validator-size` is 0 ). You can change this for your Subnet and share mempool with non Validators as well. JSON keys of these parameters are different from their matching `CLI` keys. These parameters default to the same values used for the Primary Network. For more information see [CLI Gossip Configs](#).

| CLI Key                                                 | JSON Key                               |
|---------------------------------------------------------|----------------------------------------|
| --consensus-accepted-frontier-gossip-validator-size     | gossipAcceptedFrontierValidatorSize    |
| --consensus-accepted-frontier-gossip-non-validator-size | gossipAcceptedFrontierNonValidatorSize |
| --consensus-accepted-frontier-gossip-peer-size          | gossipAcceptedFrontierPeerSize         |
| --consensus-on-accept-gossip-validator-size             | gossipOnAcceptValidatorSize            |
| --consensus-on-accept-gossip-non-validator-size         | gossipOnAcceptNonValidatorSize         |
| --consensus-on-accept-gossip-peer-size                  | gossipOnAcceptPeerSize                 |
| --consensus-app-gossip-validator-size                   | appGossipValidatorSize                 |
| --consensus-app-gossip-non-validator-size               | appGossipNonValidatorSize              |
| --consensus-app-gossip-peer-size                        | appGossipPeerSize                      |

**sidebar\_position: 2**

## **Upgrade Your AvalancheGo Node**

### **Backup Your Node**

Before upgrading your node, it is recommended you backup your staker files which are used to identify your node on the network. In the default installation, you can copy them by running following commands:

```
cd
cp ~/avalanchego/staking/staker.crt .
cp ~/avalanchego/staking/staker.key .
```

Then download `staker.crt` and `staker.key` files and keep them somewhere safe and private. If anything happens to your node or the machine node runs on, these files can be used to fully recreate your node.

If you use your node for development purposes and have keystore users on your node, you should back up those too.

## Node Installed Using the Installer Script

If you installed your node using the [installer script](#), to upgrade your node, just run the installer script again.

```
./avalanche-go-installer.sh
```

It will detect that you already have AvalancheGo installed:

```
AvalancheGo installer

Preparing environment...
Found 64bit Intel/AMD architecture...
Found AvalancheGo systemd service already installed, switching to upgrade mode.
Stopping service...
```

It will then upgrade your node to the latest version, and after it's done, start the node back up, and print out the information about the latest version:

```
Node upgraded, starting service...
New node version:
avalanche/1.1.1 [network=mainnet, database=v1.0.0, commit=f76f1fd5f99736cf468413bbac158d6626f712d2]
Done!
```

And that is it, your node is upgraded to the latest version.

If you installed your node manually, proceed with the rest of the tutorial.

## Stop the Old Node Version

After the backup is secured, you may start upgrading your node. Begin by stopping the currently running version.

### Node Running from Terminal

If your node is running in a terminal stop it by pressing `ctrl+c`.

### Node Running as a Service

If your node is running as a service, stop it by entering:

```
sudo systemctl stop avalanchego.service
```

(your service may be named differently, `avalanche.service`, or similar)

### Node Running in Background

If your node is running in the background (by running with `nohup`, for example) then find the process running the node by running `ps aux | grep avalanche`. This will produce output like:

```
ubuntu 6834 0.0 0.0 2828 676 pts/1 S+ 19:54 0:00 grep avalanche
ubuntu 2630 26.1 9.4 2459236 753316 ? S Dec02 1220:52 /home/ubuntu/build/avalanche-go
```

In this example, second line shows information about your node. Note the process id, in this case, `2630`. Stop the node by running `kill -2 2630`.

Now we are ready to download the new version of the node. You can either download the source code and then build the binary program, or you can download the pre-built binary. You don't need to do both.

Downloading pre-built binary is easier and recommended if you're just looking to run your own node and stake on it.

Building the node [from source](#) is recommended if you're a developer looking to experiment and build on Avalanche.

## Download Pre-Built Binary

If you want to download a pre-built binary instead of building it yourself, go to our [releases page](#), and select the release you want (probably the latest one.)

:::info

If you have a node, you can subscribe to the [avalanche notify service](#) with your node ID to be notified about new releases.

In addition, or if you don't have a node ID, you can get release notifications from github. To do so, you can go to our [repository](#), and look on the top-right corner for the **Watch** option. After you click on it, select **Custom**, and then **Releases**. Press **Apply** and it is done.

:::

Under `Assets`, select the appropriate file.

For MacOS:

```
Download: avalanchego-macos-<VERSION>.zip
Unzip: unzip avalanchego-macos-<VERSION>.zip
The resulting folder, avalanchego-<VERSION>, contains the binaries.
```

For Linux on PCs or cloud providers:

```
Download: avalanchego-linux-amd64-<VERSION>.tar.gz
Unzip: tar -xvf avalanchego-linux-amd64-<VERSION>.tar.gz
The resulting folder, avalanchego-<VERSION>-linux, contains the binaries.
```

For Linux on Arm64-based computers:

```
Download: avalanchego-linux-arm64-<VERSION>.tar.gz
Unzip: tar -xvf avalanchego-linux-arm64-<VERSION>.tar.gz
The resulting folder, avalanchego-<VERSION>-linux, contains the binaries.
```

You are now ready to run the new version of the node.

### Running the Node from Terminal

If you are using the pre-built binaries on MacOS:

```
./avalanchego-<VERSION>/build/avalanchego
```

If you are using the pre-built binaries on Linux:

```
./avalanchego-<VERSION>-linux/avalanchego
```

Add `nohup` at the start of the command if you want to run the node in the background.

### Running the Node as a Service

If you're running the node as a service, you need to replace the old binaries with the new ones.

```
cp -r avalanchego-<VERSION>-linux/* <DIRECTORY_WITH_OLD_BINARIES>
```

and then restart the service with `sudo systemctl start avalanchego.service`.

### Build from Source

First clone our GitHub repo (you can skip this step if you've done this before):

```
git clone https://github.com/ava-labs/avalanchego.git
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchego.git
```

You can find more about SSH and how to use it [here](#). :::

Then move to the AvalancheGo directory:

```
cd avalanchego
```

Pull the latest code:

```
git pull
```

NOTE: if the master branch has not been updated with the latest release tag, you can get to it directly via first running `git fetch --all --tags` and then `git checkout --force tags/<tag>` (where `<tag>` is the latest release tag; for example `v1.3.2`) instead of `git pull`. Note that your local copy will be in a 'detached HEAD' state, which is not an issue if you do not make changes to the source that you want push back to the repository (in which case you should check out to a branch and to the ordinary merges). Note also that the `--force` flag will disregard any local changes you might have.

Check that your local code is up to date. Do:

```
git rev-parse HEAD
```

and check that the first 7 characters printed match the Latest commit field on our [GitHub](#).

NOTE: if you used the `git checkout tags/<tag>` then these first 7 characters should match commit hash of that tag.

Now build the binary:

```
./scripts/build.sh
```

This should print:

```
Build Successful
```

You can check what version you're running by doing:

```
./build/avalanchego --version
```

You can run your node with:

```
./build/avalanchego
```

## sidebar\_position: 3

# Add a Node to the Validator Set

## Introduction

The [Primary Network](#) is inherent to the Avalanche platform and validates Avalanche's [built-in blockchains](#). In this tutorial, we'll add a node to the Primary Network on Avalanche.

The P-Chain manages metadata on Avalanche. This includes tracking which nodes are in which Subnets, which blockchains exist, and which Subnets are validating which blockchains. To add a validator, we'll issue [transactions](#) to the P-Chain.

:::warning

Note that once you issue the transaction to add a node as a validator, there is no way to change the parameters. **You can't remove your stake early or change the stake amount, node ID, or reward address.** Please make sure you're using the correct values in the API calls below. If you're not sure, feel free to join our [Discord](#) to ask questions.

:::

## Requirements

You've completed [Run an Avalanche Node](#) and are familiar with [Avalanche's architecture](#). In this tutorial, we use [AvalancheJS](#) and [Avalanche's Postman collection](#) to help us make API calls.

In order to ensure your node is well-connected, make sure that your node can receive and send TCP traffic on the staking port ( 9651 by default) and your node has a public IP address(it's optional to set --public-ip=[YOUR NODE'S PUBLIC IP HERE] when executing the AvalancheGo binary, as by default, the node will attempt to perform NAT traversal to get the node's IP according to its router). Failing to do either of these may jeopardize your staking reward.

## Add a Validator with Avalanche Wallet

First, we show you how to add your node as a validator by using [Avalanche Wallet](#).

### Retrieve the Node ID

Get your node's ID by calling [info.getNodeID](#) :

```
curl -X POST --data '{
 "jsonrpc": "2.0",
 "id": 1,
 "method": "info.getNodeID"
}' -H 'content-type:application/json' 127.0.0.1:9650/ext/info
```

The response has your node's ID:

```
{
 "jsonrpc": "2.0",
 "result": {
 "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD"
 },
 "id": 1
}
```

### Add as a Validator

Open [the wallet](#), and go the `Earn` tab. Choose `Add Validator` under the `Validate` section.

Fill out the staking parameters. They are explained in more detail in [this doc](#). When you've filled in all the staking parameters and double-checked them, click `Confirm`. Make sure the staking period is at least 2 weeks, the delegation fee rate is at least 2%, and you're staking at least 2,000 AVAX on Mainnet (1 AVAX on Fuji Testnet).

You should see a success message, and your balance should be updated.

Calling [platform.getPendingValidators](#) verifies that your transaction was accepted. Note that this API call should be made before your node's validation start time, otherwise, the return will not include your node's id as it is no longer pending.

Go back to the `Earn` tab, and click `Estimated Rewards`.

Once your validator's start time has passed, you will see the rewards it may earn, as well as its start time, end time, and the percentage of its validation period that has passed.

You can also call `platform.getCurrentValidators` to check that your node's id is included in the response.

That's it!

## Add a Validator with AvalancheJS

We can also add a node to the validator set using [AvalancheJS](#).

### Install AvalancheJS

To use AvalancheJS, you can clone the repo:

```
git clone https://github.com/ava-labs/avalanchejs.git
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchejs.git
```

You can find more about SSH and how to use it [here](#). :::

or add it to an existing project:

```
yarn add avalanche
```

For this tutorial we will use `ts-node` to run the example scripts directly from an AvalancheJS directory.

### Fuji Workflow

In this section, we will use Fuji Testnet to show how to add a node to the validator set.

Open your AvalancheJS directory and select the `examples/platformvm` folder to view the source code for the examples scripts.

We will use the `buildAddValidatorTx.ts` script to add a validator. To learn more about the `buildAddValidatorTx` API, please click [here](#).

#### Private Key

Locate this line in the file

```
const privKey: string = `${PrivateKeyPrefix}${DefaultLocalGenesisPrivateKey}`
```

and replace this with a private key that you control. You can use [this code to generate a new key](#).

```
const privKey: string = "<YOUR-PRIVATE-KEY-HERE>"
```

#### Network Setting

The following settings work when using a local node started with `--network-id=fuji`:

```
const ip: string = "localhost"
const port: number = 9650
const protocol: string = "http"
const networkID: number = 5
```

However, to connect directly to the [Avalanche Fuji Testnet API server](#), the following changes are needed:

```
const ip: string = "api.avax-test.network"
const port: number = 443
const protocol: string = "https"
const networkID: number = 5
```

Depending on the networkID passed in when instantiating an `Avalanche` object in the code, the encoded addresses used will have a distinctive Human Readable Part(HRP) per network.

Example Address: 5 - X- `fuji 19rk...zrlu33yxqzg0h`

For Fuji Testnet, 5 is the correct value to use.

To learn more about encoded addresses, click [here](#).

#### Settings for Validation

Next we need to specify the node's validation period and delegation fee.

```
const nodeID: string = "NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg"
const startTime: BN = UnixNow().add(new BN(60 * 1))
const endTime: BN = startTime.add(new BN(26300000))
const delegationFee: number = 10
```

## Node ID

This is the node ID of the validator being added. See [above section](#) on how to retrieve the node id by using API [info.getNodeID](#).

## Staking Period

`startTime` and `endTime` are required to specify the time of starting/leaving validation. The minimum duration that one can validate the Primary Network is 2 weeks, and the maximum duration is one year. One can start a new validation on the Primary Network after finishing one, it's just that the maximum *continuous* duration is one year. `startTime` and `endTime` are the Unix times when your validator will start and stop validating the Primary Network, respectively. `startTime` must be in the future relative to the time the transaction is issued.

The sample code uses `const startTime: BN = UnixNow().add(new BN(60 * 1))` and `const endTime: BN = startTime.add(new BN(26300000))` to compute the Unix time 1 minute and 304 days in the future (at the time when this article was written) to use as the values of `startTime` and `endTime`, respectively.

:::tip

You can create your own Unix timestamp [here](#) or by using the `UnixNow()` method

:::

To create your own start times, please follow the steps below:

Locate this line in the file

```
const startTime: BN = UnixNow().add(new BN(60 * 1))
const endTime: BN = startTime.add(new BN(26300000))
```

Change `startTime` and `endTime` to new `BN` values, for example:

```
const startTime: BN = new BN(1654656829) // Wed Jun 08 2022 02:53:49 GMT+0000
const endTime: BN = new BN(1662602029) // Thu Sep 08 2022 01:53:49 GMT+0000
```

## Delegation Fee Rate

Avalanche allows for delegation of stake. This parameter is the percent fee this validator charges when others delegate stake to them. For example, if `delegationFeeRate` is `10` and someone delegates to this validator, then when the delegation period is over, 10% of the reward goes to the validator and the rest goes to the delegator, if this node meets the validation reward requirements.

## Stake Amount

Set the proper staking amount in calling `pchain.buildAddValidatorTx` by replacing `stakeAmount.minValidatorStake` with a number in the unit of gwei, for example, `BN(1e12)` which is 10,000 AVAX.

## Addresses

By default, the example uses the variable `pAddressStrings` to define `toAddresses`, `fromAddresses`, `changeAddresses` and `rewardAddresses`:

```
const pAddressStrings: string[] = pchain.keyChain().getAddressStrings()
```

This retrieves the P-Chain addresses that belong to the `private key` that appears earlier in the example.

No change is needed in the addresses for the default action. For customization, please refer to [this section](#).

## Execute the Code

Now that we have made all of the necessary changes to the example script, it's time to add a validator to the Fuji Network.

Run the command:

```
ts-node examples/platformvm/buildAddValidatorTx.ts
```

The response has the transaction ID.

```
Success! TXID: 2ftDVwmss5eJk8HFsvNVi6a3vWK9s3szZFhEeSY2HCS8xD8Cra
```

We can check the transaction's status by running the example script: [getTxStatus.ts](#) following the steps below:

1. Ensure that your [network settings](#) are correct before running the script.

2. Locate this line in the file

```

const main = async () : Promise<any> => {
 const txID: string = "x1NLb9JaHkKTxvSRReVssFwQ38mY7bfD1Ky1BPv721VhrpuSE"
 ...
}

```

and replace it with the `buildAddValidator` TXID

```

const main = async () : Promise<any> => {
 const txID: string = "2ftDVwmss5eJk8HFsNVi6a3vWK9s3zzFhEeSY2HCS8xDb8Cra"
 ...
}

```

Run the command:

```
ts-node examples/platformvm/getTxStatus.ts
```

This returns:

```
{ status: 'Committed' }
```

The status should be `Committed`, meaning the transaction was successful.

We can see if the node is now in the pending validator set for the Fuji network by using the example: [getPendingValidators.ts](#). Just change the [network settings](#) to meet Fuji requirements and then run the script:

```
ts-node examples/platformvm/getPendingValidators.ts
```

The response should include the node we just added:

```
{
 "validators": [
 {
 "nodeID": "NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg",
 "startTime": "1654656829",
 "endTime": "1662602029",
 "stakeAmount": "1000000000"
 }
],
 "delegators": []
}
```

When the time reaches `1654656829` (Wed Jun 08 2022 02:53:49 GMT+0000), this node will start validating the Primary Network. When it reaches `1662602029` (Thu Sep 08 2022 01:53:49 GMT+0000), this node will stop validating the Primary Network. The staked AVAX and the rewards, if any, will be returned to `pAddressStrings`.

### Customizing Addresses

There are 4 addresses which are needed when calling `pchain.buildAddValidatorTx`. Only 2 of them can be changed: `toAddresses` and `rewardAddresses`. For backward-compatibility reasons, `fromAddresses` and `changeAddresses` are just placeholders and are ignored.

`toAddresses`

An array of addresses who receive the staked tokens at the end of the staking period.

`rewardAddresses`

When a validator stops validating the Primary Network, they will receive a reward if they are sufficiently responsive and correct while they validated the Primary Network. These tokens are sent to `rewardAddresses`. The original stake will be sent back to the addresses defined in `toAddresses`.

A validator's stake is never slashed, regardless of their behavior they will always receive their stake back when they're done validating.

Locate this part of the code

```

let privKey: string = `${PrivateKeyPrefix}${DefaultLocalGenesisPrivateKey}`
pKeychain.importKey(privKey)

```

and replace `privKey` with private keys that you control. To generate a new keypair, we can use the [createKeypair.ts](#) example script along with [Fuji Network Settings](#).

```

let privKey: string =
 "PrivateKey-PY2dVfxzvBAela5nn7x23wmZMgAYJaS3XAZXzdUa22JtzUvKM"
pKeychain.importKey(privKey)
privKey = "PrivateKey-2Y3Vg9LShMJyUDBHxQqv5WtKDj8yAVHyM3H5CNCCBmtg3pQEQQ"
pKeychain.importKey(privKey)
privKey = "PrivateKey-NaV16owRSfa5TAtxtoU1BPu0M2y1ohttRbwKJG1j7onE4Ge1s"

```

```

pKeychain.importKey(privKey)
priKey = "PrivateKey-26JMUsR5RCKf5k9ME8WxKCWEuCK5s2SrALUn7vEa2urwyDDc91"
pKeychain.importKey(privKey)

const pAddressStrings: string[] = pchain.keyChain().getAddressStrings()

```

This example would create a keychain with 4 addresses:

```

"P-fuji1jx644d9y00y5q4hz8cq4wr75a2erne2y4e32xc", // pAddressStrings[0]
"P-fuji1wchdgd94j8szlpsp56gvgkvdn20svpmnm8qk", // pAddressStrings[1]
"P-fuji1f36kkpy6yzd7ayrywxvvprns7qlrcu3hwqdy8", // pAddressStrings[2]
"P-fuji1qw7yt3fp43kuwsuff4vhezs2y100s1r09vmhs", // pAddressStrings[3]

```

Now we can pass in each address according to its slot in the `pAddressStrings` array:

```

const unsignedTx: UnsignedTx = await pchain.buildAddValidatorTx(
 utxoSet,
 [pAddressStrings[0], pAddressStrings[1]], // toAddresses, one or more addresses
 [pAddressStrings[0]], // fromAddresses, required for backward-compatibility
 [pAddressStrings[0]], // changeAddresses, required for backward-compatibility
 nodeID,
 startTime,
 endTime,
 stakeAmount.minValidatorStake,
 [pAddressStrings[2], pAddressStrings[3]], // rewardAddresses, one or more addresses
 delegationFee,
 locktime,
 threshold,
 memo,
 asOf
)

```

## Mainnet Workflow

The Fuji workflow above can be adapted to Mainnet with the following modifications:

- The correct private key.
- Network setting should be to a Mainnet node, either [a local node on Mainnet](#) or [Avalanche Mainnet API server](#) where `api.avax.network` should be used for the `ip`.
- `const networkID: number = 1` based on [this](#).
- Set the correct amount to stake.

## **sidebar\_position: 1 description: Learn how to stake on Avalanche by validating or delegating**

### What Is Staking?

Staking is the process of locking up tokens to support a network while receiving a reward in return (rewards can be increased network utility, monetary compensation, etc.). The concept of staking was [first formally introduced](#) by Sunny King and Scott Nadal of Peercoin.

### How Does Proof-of-Stake Work?

To resist [sybil attacks](#), a decentralized network must require that network influence is paid with a scarce resource. This makes it infeasible expensive for an attacker to gain enough influence over the network to compromise its security. In proof-of-work systems, the scarce resource is computing power. On Avalanche, the scarce resource is the native token, [AVAX](#). For a node to [validate](#) a blockchain on Avalanche, it must stake AVAX.

### Staking Parameters on Avalanche

When a validator is done validating the [Primary Network](#), it receives back the AVAX tokens it staked. It may receive a reward for helping to secure the network. A validator only receives a [validation reward](#) if it is sufficiently responsive and correct during the time it validates. Read the [Avalanche token white paper](#) to learn more about AVAX and the mechanics of staking.

:::caution

Staking rewards are sent to your wallet address at the end of the staking term **as long as all of these parameters are met**.

:::

### Mainnet

- The minimum amount that a validator must stake is 2,000 AVAX
- The minimum amount that a delegator must delegate is 25 AVAX
- The minimum amount of time one can stake funds for validation is 2 weeks
- The maximum amount of time one can stake funds for validation is 1 year
- The minimum amount of time one can stake funds for delegation is 2 weeks
- The maximum amount of time one can stake funds for delegation is 1 year
- The minimum delegation fee rate is 2%

- The maximum weight of a validator (their own stake + stake delegated to them) is the minimum of 3 million AVAX and 5 times the amount the validator staked. For example, if you staked 2,000 AVAX to become a validator, only 8000 AVAX can be delegated to your node total (not per delegator)

A validator will receive a staking reward if they are online and responsive for more than 80% of their validation period, as measured by a majority of validators, weighted by stake. **You should aim for your validator to be online and responsive 100% of the time.**

You can call API method `info.uptime` on your node to learn its weighted uptime and what percentage of the network currently thinks your node has an uptime high enough to receive a staking reward. See [here](#). You can get another opinion on your node's uptime from Avalanche's [Validator Health dashboard](#). If your reported uptime is not close to 100%, there may be something wrong with your node setup, which may jeopardize your staking reward. If this is the case, please see [here](#) or contact us on [Discord](#) so we can help you find the issue. Note that only checking the uptime of your validator as measured by non-staking nodes, validators with small stake, or validators that have not been online for the full duration of your validation period can provide an inaccurate view of your node's true uptime.

## Fuji Testnet

On Fuji Testnet, all staking parameters are the same as those on Mainnet except the following ones:

- The minimum amount that a validator must stake is 1 AVAX
- The minimum amount that a delegator must delegate is 1 AVAX
- The minimum amount of time one can stake funds for validation is 24 hours
- The minimum amount of time one can stake funds for delegation is 24 hours

## Reward Formula

Consider a validator which stakes a  $\$Stake$  amount of Avax for  $\$StakingPeriod$  seconds.

Assume that at the start of the staking period there is a  $\$Supply$  amount of Avax in the Primary Network. The maximum amount of Avax is  $\$MaximumSupply$ .

Then at the end of its staking period, a responsive validator receives a reward calculated as follows:

$$\begin{aligned} \text{\$\$ Reward} &= (\text{MaximumSupply} - \text{Supply}) \times \frac{\text{Stake}}{\text{Supply}} \times \frac{\text{Staking Period}}{\text{Minting Period}} \times \text{EffectiveConsumptionRate} \\ \text{where } \text{\$\$ EffectiveConsumptionRate} &= \text{\$\$} \times \frac{\text{MinConsumptionRate}}{\text{PercentDenominator}} \times \left(1 - \frac{\text{Staking Period}}{\text{Minting Period}}\right) + \frac{\text{MaxConsumptionRate}}{\text{PercentDenominator}} \times \frac{\text{Staking Period}}{\text{Minting Period}} \end{aligned}$$

Note that  $\$StakingPeriod$  is the staker's entire staking period, not just the staker's uptime, that is the aggregated time during which the staker has been responsive. The uptime comes into play only to decide whether a staker should be rewarded; to calculate the actual reward, only the staking period duration is taken into account.

$\$EffectiveConsumptionRate$  is a linear combination of  $\$MinConsumptionRate$  and  $\$MaxConsumptionRate$ .  $\$MinConsumptionRate$  and  $\$MaxConsumptionRate$  bound  $\$EffectiveConsumptionRate$  because

$\$MinConsumptionRate \leq \text{EffectiveConsumptionRate} \leq \text{MaxConsumptionRate}$

The larger  $\$StakingPeriod$  is, the closer  $\$EffectiveConsumptionRate$  is to  $\$MaxConsumptionRate$ .

A staker achieves the maximum reward for its stake if  $\$StakingPeriod = \$Minting Period$ . The reward is:

$$\text{\$\$ Max Reward} = (\text{MaximumSupply} - \text{Supply}) \times \frac{\text{Stake}}{\text{Supply}} \times \frac{\text{MaxConsumptionRate}}{\text{PercentDenominator}}$$

## Validators

Validators secure Avalanche, create new blocks/vertices, and process transactions. To achieve consensus, validators repeatedly sample each other. The probability that a given validator is sampled is proportional to its stake.

When you add a node to the validator set, you specify:

- Your node's ID
- When you want to start and stop validating
- How many AVAX you are staking
- The address to send any rewards to
- Your delegation fee rate (see below)

:::info The minimum amount that a validator must stake is 2,000 AVAX. :::

:::warning

Note that once you issue the transaction to add a node as a validator, there is no way to change the parameters. **You can't remove your stake early or change the stake amount, node ID, or reward address.** Please make sure you're using the correct values in the API calls below. If you're not sure, ask for help on [Discord](#). If you want to add more tokens to your own validator, you can delegate the tokens to this node - but you cannot increase the base validation amount (so delegating to yourself goes against your delegation cap).

:::

## Running a Validator

If you're running a validator, it's important that your node is well connected to ensure that you receive a reward.

When you issue the transaction to add a validator, the staked tokens and transaction fee (which is 0) are deducted from the addresses you control. When you are done validating, the staked funds are returned to the addresses they came from. If you earned a reward, it is sent to the address you specified when you added yourself as a validator.

### Allow API Calls

To make API calls to your node from remote machines, allow traffic on the API port ( 9650 by default), and run your node with argument `--http-host=`

You should disable all APIs you will not use via command-line arguments. You should configure your network to only allow access to the API port from trusted machines (for example, your personal computer.)

#### Why Is My Uptime Low?

Every validator on Avalanche keeps track of the uptime of other validators. Every validator has a weight (that is the amount staked on it.) The more weight a validator has, the more influence they have when validators vote on whether your node should receive a staking reward. You can call API method `info.uptime` on your node to learn its weighted uptime and what percentage of the network stake currently thinks your node has an uptime high enough to receive a staking reward.

You can also see the connections a node has by calling `info.peers`, as well as the uptime of each connection. **This is only one node's point of view.** Other nodes may perceive the uptime of your node differently. Just because one node perceives your uptime as being low does not mean that you will not receive staking rewards.

If your node's uptime is low, make sure you're setting config option `--public-ip=[NODE'S PUBLIC IP]` and that your node can receive incoming TCP traffic on port 9651.

#### Secret Management

The only secret that you need on your validating node is its Staking Key, the TLS key that determines your node's ID. The first time you start a node, the Staking Key is created and put in `$HOME/.avalanchego/staking/staker.key`. You should back up this file (and `staker.crt`) somewhere secure. Losing your Staking Key could jeopardize your validation reward, as your node will have a new ID.

You do not need to have AVAX funds on your validating node. In fact, it's best practice to **not** have a lot of funds on your node. Almost all of your funds should be in "cold" addresses whose private key is not on any computer.

#### Monitoring

Follow this [tutorial](#) to learn how to monitor your node's uptime, general health, etc.

## Delegators

A delegator is a token holder, who wants to participate in staking, but chooses to trust an existing validating node through delegation.

When you delegate stake to a validator, you specify:

- The ID of the node you're delegating to
- When you want to start/stop delegating stake (must be while the validator is validating)
- How many AVAX you are staking
- The address to send any rewards to

:::info The minimum amount that a delegator must delegate is 25 AVAX. :::

:::warning

Note that once you issue the transaction to add your stake to a delegator, there is no way to change the parameters. **You can't remove your stake early or change the stake amount, node ID, or reward address.** If you're not sure, ask for help on [Discord](#).

:::

#### Delegator Rewards

If the validator that you delegate tokens to is sufficiently correct and responsive, you will receive a reward when you are done delegating. Delegators are rewarded according to the same function as validators. However, the validator that you delegate to keeps a portion of your reward specified by the validator's delegation fee rate.

When you issue the transaction to delegate tokens, the staked tokens and transaction fee are deducted from the addresses you control. When you are done delegating, the staked tokens are returned to your address. If you earned a reward, it is sent to the address you specified when you delegated tokens.

## FAQ

### Is There a Tool to Check the Health of a Validator?

Yes, just enter your node's ID in the Avalanche Stats [Validator Health Dashboard](#).

### How Is It Determined Whether a Validator Receives a Staking Reward?

When a node leaves the validator set, the validators vote on whether the leaving node should receive a staking reward or not. If a validator calculates that the leaving node was responsive for more than the required uptime (currently 80%), the validator will vote for the leaving node to receive a staking reward. Otherwise, the validator will vote that the leaving node should not receive a staking reward. The result of this vote, which is weighted by stake, determines whether the leaving node receives a reward or not.

Each validator only votes "yes" or "no." It does not share its data such as the leaving node's uptime.

Each validation period is considered separately. That is, suppose a node joins the validator set, and then leaves. Then it joins and leaves again. The node's uptime during its first period in the validator set does not affect the uptime calculation in the second period, hence, has no impact on whether the node receives a staking reward for its second period in the validator set.

### How Are Delegation Fees Distributed To Validators?

If a validator is online for 80% of a delegation period, they receive a % of the reward (the fee) earned by the delegator. The P-Chain used to distribute this fee as a separate UTXO per delegation period. After the [Cortina Activation](#), instead of sending a fee UTXO for each successful delegation period, fees are now batched during a node's entire validation period and are distributed when it is unstaked.

### Error: Couldn't Issue TX: Validator Would Be Over Delegated

This error occurs whenever the delegator can not delegate to the named validator. This can be caused by the following.

- The delegator `startTime` is before the validator `startTime`
- The delegator `endTime` is after the validator `endTime`
- The delegator weight would result in the validator total weight exceeding its maximum weight

## Quick Start Overview

| Title                                                                   | Description                                                              |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------|
| <a href="#">Fuji Workflow</a>                                           | Avalanche Fuji Workflow.                                                 |
| <a href="#">Transfers AVAX Tokens Between Chains</a>                    | Send AVAX between Chains                                                 |
| <a href="#">Multi Signature UTXOs with AvalancheJS</a>                  | Learn about creating and issuing multi signature UTXOs with AvalancheJS  |
| <a href="#">Avalanche Transaction Fee</a>                               | Avalanche Transaction Fee for all chains                                 |
| <a href="#">Adjusting Gas Price During High Network Activity</a>        | Adjusting Gas Price During High Network Activity                         |
| <a href="#">Sending Transactions with Dynamic Fees using JavaScript</a> | Sending Transactions with Dynamic Fees using JavaScript                  |
| <a href="#">C-Chain Exchange Integration</a>                            | Guidelines for exchange integration with Ethereum-compatible C-Chain     |
| <a href="#">Tools and Utilities</a>                                     | Collection of popular tools for maintaining nodes and developing Subnets |
| <a href="#">Flow of a Single Blockchain</a>                             | The different components of a blockchain                                 |

**description:** This tutorial will help users to adjust their priority fee and max fee cap during high network activity and take advantage of the benefits of dynamic fee transactions.

## Adjusting Gas Price During High Network Activity

Sometimes during periods of high network activity, transactions either remain pending for a very long duration or instantly get a failed transaction notification. This may confuse and frustrate users, especially if they don't understand why their transactions are not getting accepted.

### Probable Reasons You Are Here

- Your transaction has stalled, and you don't know what to do
- Your transaction has failed, with an error - `transaction underpriced`
- It's your first transaction, and you want to be sure about any potential issues
- Just for general knowledge on adjusting dynamic fee settings

If these are your reasons for being here, then you can either go through this entire section, for a better understanding of the scenario or directly skip to the [solution](#).

### Good to Know Keywords and Concepts

The amount of computation used by a transaction is measured in units of `gas`. Each unit of gas is paid for in AVAX at the `gas price` for the transaction. The `gas price` of the transaction is determined by the parameters of the transaction and the `base fee` of the block that it is included in.

To avoid draining the user's wallet due to non-terminating execution through the EVM, transactions are submitted with a `gas limit`, which denotes the maximum units of gas that a particular transaction is allowed to consume.

If a transaction attempts to use more than this limit, then the transaction will revert and still consume and pay for the full `gas limit`. Total fees paid by the user can be calculated as `(gas consumed) * (gas price)`, and is known as `gas fees`. Similarly, maximum gas fees can be calculated as `(gas limit) * (gas price)`.

Originally, transactions could only set a single parameter to define how much they were willing to pay for gas: `gas price`. When dynamic fees were introduced, EIP-1559 style transactions were introduced as well which contain two parameters `maxFeeCap` and `maxPriorityFee` to determine the price a transaction is willing to pay.

With the introduction of dynamic fees, legacy style transactions that only have a single `gas price` parameter can lead to both delayed transactions and overpaying for transactions. Dynamic fee transactions are the solution! For more info, read [this](#).

For the dynamic fee algorithm, when a block is produced or verified, we look over the past 10s to see how much gas has been consumed within that window (with an added charge for each block produced in that window) to determine the current network utilization. This window has a target utilization, which is currently set to `15M` gas units. Lastly, there is an added charge if a block is produced faster than the target rate of block production. Currently, the target rate of block production is one block every two seconds, so if a new block is produced one second after its parent, then there is an additional surcharge added into the base fee calculation.

Base price could increase, decrease, or remain the same depending upon the amount of activity on the network in the most recent window. If the total gas in the last few blocks of the window is more, less or the same than the target gas, then the base price will increase, decrease, or remain the same, respectively.

When estimating the base fee for users, we simply look at the currently preferred block and calculate what the base fee would be for a block built on top of that block immediately.

Along with a gas limit, users can now pass 2 values in dynamic fee transactions

- `gas fee cap` and `gas tip cap`.

The maximum price per unit of gas, that the user is willing to pay for their transaction is called `gas fee cap`. If the base price for a block is more than the gas fee cap, then the transaction will remain in the transaction pool until the base fee has been changed to be less than or equal to the provided gas fee cap (note: the transaction pool limits the number of pending transactions, so if the number of pending transactions exceeds the configured cap then the transactions with the lowest fees may be evicted from the transaction pool and need to be re-issued).

`Gas tip cap` is the maximum price per unit of gas, that the user is willing to pay above the base price to prioritize their transaction. But the tip is capped by both the `gas tip cap` as well as the `gas fee cap`. The actual tip paid above the `base fee` of the block is known as the `effective gas tip`.

```
EffectiveTip = min(MaxFeeCap - BaseFee, GasTipCap)
```

Consider the following examples (here Gwei or nAVAX is one-billionth of AVAX) –

| Transaction | Max Fee Cap | Gas tip cap | Base price | Effective tip | Total price |
|-------------|-------------|-------------|------------|---------------|-------------|
| 1           | 50 Gwei     | 0 Gwei      | 25 Gwei    | 0 Gwei        | 25 Gwei     |
| 2           | 50 Gwei     | 0 Gwei      | 50 Gwei    | 0 Gwei        | 50 Gwei     |
| 3           | 50 Gwei     | 0 Gwei      | 60 Gwei    | 0 Gwei        | PENDING     |
| A           | 50 Gwei     | 10 Gwei     | 40 Gwei    | 10 Gwei       | 50 Gwei     |
| B           | 40 Gwei     | 40 Gwei     | 40 Gwei    | 0 Gwei        | 40 Gwei     |

Look at transactions **A** and **B** (the bottom two transactions). In these scenarios, it looks like transaction B is paying a higher tip, however, this depends on the base fee of the block where the transactions are included. The effective tip of A is more than that of B. So, if both of these transaction competes for being included in the next block, then the validators would prioritize transaction A since it pays a higher effective tip.

## Why My Transaction is on Hold or Failing?

If your transaction is failing and giving an error - `transaction underpriced`, then the `max fee cap` of your transaction must be less than the minimum base price that the network supports (as of now, it's 25 nAVAX or Gwei). Although the `base fee` is automatically estimated in wallets like MetaMask, you can try increasing the `max fee cap` in the wallet.

During a period of heavy congestion on the network, all submitted transactions can't be included in the same block, due to the block's gas limit. So, validators choose transactions giving higher priority to transactions with the highest effective tips.

Another reason your transaction may get stuck in pending, is that the `max fee cap` may be below the current base fee that the network is charging. In this case, you need to increase the `max fee cap` of your transaction above the current base fee for it to be included in the block.

These fee adjustments can be made through wallets like MetaMask.

## Adjusting Gas Fees Before Submitting the Transaction

You may not need to edit the gas fees on normal days. This is only required if there is heavy congestion on the network, and the base fees are frequently fluctuating.

1. Let's create a sample transaction on Avalanche Mainnet, in which we will be sending 0.1 AVAX to a receiver using the MetaMask. By clicking **Next** we can review gas fees and the amount which we want to send.



2. On the review page, you can see the estimated priority and max fee for this transaction. Now click on **EDIT**, to adjust these fees according to network requirements.



3. On this page, you can edit the priority fee (`gas tip cap`) and max fee (`max fee cap`) using the advanced options or you can use the MetaMask's inbuilt gas estimation. You can estimate the max fee as shown on [Snowtrace](#) which represents the average max fee over the last 3 seconds. For more detailed statistics, you can have a look [here](#).



4. If the network activity is high, you have to edit the priority and max fees accordingly, as given on Snowtrace. Consider the example below, where the average max fee is 30 Gwei (nAVAX).



5. It is recommended to set the `max fee cap` as the maximum price that you are willing to pay for a transaction, no matter how high or low the `base fee` will be, as you will only be charged the minimum of `base fee` and the `max fee cap`, along with a small priority fee above the `base fee`. Apart from that, MetaMask will indicate you whether you are paying extra price than required, so you should not have to worry about getting overcharged. Now let's edit the `max fee` to 40 Gwei. This would ensure that our transaction would not fail until the `base fee` would exceed this amount. We can set a priority fee to

anything between 0 and 40 Gwei. More the priority fee faster will be the transaction. For this example, let's set this to 2 Gwei. However you can always rely on MetaMask's automatic estimate instead of opting for advanced options. Now, save and confirm the transaction.



6. After submitting the transaction, even if the base fee has decreased, you will only pay 2 Gwei above that fee as a priority fee. If this fee is one of the highest among the pending transactions, then it will be confirmed rapidly. We can see the confirmation of the transaction below.



| Transaction | Max Fee Cap | Gas tip cap | Base price | Effective tip | Total price |
|-------------|-------------|-------------|------------|---------------|-------------|
| 1           | 40 Gwei     | 2 Gwei      | 25 Gwei    | 2 Gwei        | 27 Gwei     |

## Speeding Up the Pending Transaction

If your transaction is on hold for a very long time, you can speed up through MetaMask. As shown in the below image, click on the **Speed Up** button, to edit your priority and max fee. By default, the new transaction has slightly more priority and max fee (say 10% more than the previous), but you can edit as per your convenience.



You can look at Snowtrace's gas tracker to get an estimate about the average base fee, which is currently getting accepted. If the max fee of the network is much higher than your transaction, then set the max fee cap to match that of the network.

## Flow of a Single Blockchain



### Intro

The Avalanche network consists of 3 built-in blockchains: X-Chain, C-Chain, and P-Chain. The X-Chain is used to manage assets and uses the Avalanche consensus protocol. The C-Chain is used to create and interact with smart contracts and uses the Snowman consensus protocol. The P-Chain is used to coordinate validators and stake and also uses the Snowman consensus protocol. At the time of writing, the Avalanche network has ~1200 validators. A set of validators makes up a Subnet. Subnets can validate 1 or more chains. It is a common misconception that 1 Subnet = 1 chain and this is shown by the primary Subnet of Avalanche which is made up of the X-Chain, C-Chain, and P-Chain.

A node in the Avalanche network can either be a validator or a non-validator. A validator stakes AVAX tokens and participates in consensus to earn rewards. A non-validator does not participate in consensus or have any AVAX staked but can be used as an API server. Both validators and non Validators need to have their own copy of the chain and need to know the current state of the network. At the time of writing, there are ~1200 validators and ~1800 non-validators.

Each blockchain on Avalanche has several components: the virtual machine, database, consensus engine, sender, and handler. These components help the chain run smoothly. Blockchains also interact with the P2P layer and the chain router to send and receive messages.

## Peer-to-Peer (P2P)

### Outbound Messages

[The OutboundMsgBuilder interface](#) specifies methods that build messages of type `OutboundMessage`. Nodes communicate to other nodes by sending `OutboundMessage` messages.

All messaging functions in `OutboundMsgBuilder` can be categorized as follows:

- **Handshake**
  - Nodes need to be on a certain version before they can be accepted into the network.
- **State Sync**
  - A new node can ask other nodes for the current state of the network. It only syncs the required state for a specific block.
- **Bootstrapping**
  - Nodes can ask other nodes for blocks to build their own copy of the chain. A node can fetch all blocks from the locally last accepted block to the current last accepted block in the network.
- **Consensus**
  - Once a node is up to tip they can participate in consensus! During consensus, a node conducts a poll to several different small random samples of the validator set. They can communicate decisions on whether or not they have accepted/rejected a block.
- **App**
  - VMs communicate application-specific messages to other nodes through app messages. A common example is mempool gossiping.

Currently, AvalancheGo implements its own message serialization to communicate. In the future, AvalancheGo will use protocol buffers to communicate.

### Network

[The networking interface](#) is shared across all chains. It implements functions from the `ExternalSender` interface. The two functions it implements are `Send` and `Gossip`. `Send` sends a message of type `OutboundMessage` to a specific set of nodes (specified by an array of `NodeIDs`). `Gossip` sends a message of

type `OutboundMessage` to a random group of nodes in a Subnet (can be a validator or a non-validator). Gossiping is used to push transactions across the network. The networking protocol uses TLS to pass messages between peers.

Along with sending and gossiping, the networking library is also responsible for making connections and maintaining connections. Any node, either a validator or non-validator, will attempt to connect to the primary network.

## Router

The `ChainRouter` routes all incoming messages to its respective blockchain using `ChainID`. It does this by pushing all the messages onto the respective Chain handler's queue. The `ChainRouter` references all existing chains on the network such as the X-chain, C-chain, P-chain and possibly any other chain. The `ChainRouter` handles timeouts as well. When sending messages on the P2P layer, timeouts are registered on the sender and cleared on the `ChainRouter` side when a response is received. If no response is received, then it triggers a timeout. Because timeouts are handled on the `ChainRouter` side, the handler is reliable. Timeouts are triggered when peers do not respond and the `ChainRouter` will still notify the handler of failure cases. The timeout manager within `ChainRouter` is also adaptive. If the network is experiencing long latencies, timeouts will then be adjusted as well.

## Handler

The main function of `the Handler` is to pass messages from the network to the consensus engine. It receives these messages from the `ChainRouter`. It passes messages by pushing them onto a sync or Async queue (depends on message type). Messages are then popped from the queue, parsed, and routed to the correct function in consensus engine. This can be one of the following.

- State sync message (sync queue)
- Bootstrapping message (sync queue)
- Consensus message (sync queue)
- App message (Async queue)

## Sender

The main role of `the sender` is to build and send outbound messages. It is actually a very thin wrapper around the normal networking code. The main difference here is that sender registers timeouts and tells the router to expect a response message. The timer starts on the sender side. If there is no response, sender will send a failed response to the router. If a node is repeatedly unresponsive, that node will get benched and the sender will immediately start marking those messages as failed. If a sufficient amount of network deems the node benched, it might not get rewards (as a validator).

## Consensus Engine

Consensus is defined as getting a group of distributed systems to agree on an outcome. In the case of the Avalanche network, consensus is achieved when validators are in agreement with the state of the blockchain. The novel consensus algorithm is documented in the [white paper](#). There are two main consensus algorithms: Avalanche and [Snowman](#). The engine is responsible for adding proposing a new block to consensus, repeatedly polling the network for decisions (accept/reject), and communicating that decision to the `Sender`.

## Blockchain Creation

The `Manager` is what kick-starts everything in regards to blockchain creation, starting with the P-Chain. Once the P-Chain finishes bootstrapping, it will kickstart C-Chain and X-Chain and any other chains. The `Manager`'s job is not done yet, if a create-chain transaction is seen by a validator, a whole new process to create a chain will be started by the `Manager`. This can happen dynamically, long after the 3 chains in the Primary Network have been created and bootstrapped.

# Transfer AVAX Tokens Between Chains

## Introduction

This article shows how to transfer AVAX tokens programmatically between any two chains (X/P/C chains) of the Primary Network.

If you are looking for how to transfer AVAX tokens using the web wallet, please check out [this article](#).

## Prerequisites

- You are familiar with [Avalanche's architecture](#).
- You have completed [Run an Avalanche Node](#).
- You are familiar with [AvalancheJS](#).
- You have installed [ts-node](#) so that you can follow examples in this tutorial.

## Getting Started

To use AvalancheJS, you can clone the repo:

```
git clone https://github.com/ava-labs/avalanchejs.git
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchejs.git
```

You can find more about SSH and how to use it [here](#). :::

or add it to an existing project:

```
yarn add --dev avalanche
```

For this tutorial we will use `ts-node` to run the example scripts directly from an AvalancheJS directory.

In order to send AVAX, you need to have some AVAX. You can use a pre-funded account on local network or get testnet AVAX from the [Avalanche Faucet](#), which is an easy way to get to play around with Avalanche. After getting comfortable with your code, you can run the code on Mainnet after making necessary changes.

## Transferring AVAX Using AvalancheJS

The easiest way to transfer AVAX between chains is to use [AvalancheJS](#) which is a programmatic way to access and move AVAX.

AvalancheJS allows you to create and sign transactions locally which is why it is the recommended way to transfer AVAX between chains. We are moving away from using AvalancheGo's keystore because it requires you to keep your keys on a full node which makes them a target for malicious hackers.

### Example Code

Following files can be found under the [examples](#) directory of the AvalancheJS project.

| Transfer From >> To | Export                                           | Import                                             |
|---------------------|--------------------------------------------------|----------------------------------------------------|
| X-Chain >> C-Chain  | <a href="#">X-Chain : Export Avax to C-Chain</a> | <a href="#">C-Chain : Import Avax from X-Chain</a> |
| X-Chain >> P-Chain  | <a href="#">X-Chain : Export Avax to P-Chain</a> | <a href="#">P-Chain : Import Avax from X-Chain</a> |
| P-Chain >> X-Chain  | <a href="#">P-Chain : Export Avax to X-Chain</a> | <a href="#">X-Chain : Import Avax from P-Chain</a> |
| P-Chain >> C-Chain  | <a href="#">P-Chain : Export Avax to C-Chain</a> | <a href="#">C-Chain : Import Avax from P-Chain</a> |
| C-Chain >> X-Chain  | <a href="#">C-Chain : Export Avax to X-Chain</a> | <a href="#">X-Chain : Import Avax from C-Chain</a> |
| C-Chain >> P-Chain  | <a href="#">C-Chain : Export Avax to P-Chain</a> | <a href="#">P-Chain : Import Avax from C-Chain</a> |

:::tip

The naming convention in the file and directory names are:

AVM is for X-Chain, EVM for C-Chain, and PlatformVM for P-Chain.

:::

### Transaction Fee

Transaction fees are fixed on X-Chain and P-Chain, while dynamic on C-Chain, see [this article](#) for details. When transferring tokens, please take fee into consideration in calculating total amount to be transferred.

## Fuji Workflow

This tutorial uses [X-Chain <-> C-Chain](#) transfers as an example. Transferring between other chains are very similar.

### Transfer from the X-Chain to the C-Chain

To transfer a specified amount token from X-Chain to C-Chain, the token needs to be first exported from the X-Chain to the atomic memory, from where it is then imported to C-Chain.

#### Export the Avax Token From X-Chain to C-Chain

Select the `examples/avm` folder to view the AvalancheJS X-Chain examples. To export AVAX from the X-Chain to the C-Chain, select [avm/buildExportTx-cchain-avax.ts](#).

#### Private Key

Locate this line in the file

```
const privKey: string = `${PrivateKeyPrefix}${DefaultLocalGenesisPrivateKey}`
```

and replace this with a private key that you control.

```
const privKey: string = "<YOUR-PRIVATE-KEY-HERE>"
```

#### Network Setting

The following settings work when using a local node started with [--network-id=fuji](#):

```
const ip: string = "localhost"
const port: number = 9650
const protocol: string = "http"
const networkID: number = 5
```

However, to connect directly to the [Avalanche Fuji Testnet API server](#), the following changes are needed:

```
const ip: string = "api.avax-test.network"
const port: number = 443
const protocol: string = "https"
const networkID: number = 5
```

Depending on the networkID passed in when instantiating Avalanche, the encoded addresses used will have a distinctive Human Readable Part(HRP) per each network.

Example Address: 5 - X- fuji 19rknw8l0grnfunjrzwxlxync6zrlu33yxqzg0h

For Fuji Testnet, 5 is the correct value to use.

```
const networkID: number = 5
```

To learn more about encoded addresses, click [here](#).

#### Set the Correct Amount To Send:

By default the scripts send the wallet's entire AVAX balance:

```
const balance: BN = new BN(getBalanceResponse.balance)
const amount: BN = balance.sub(fee)
```

To send a different amount, please replace the code above with the following. Below sets a new value of 0.01 AVAX ( 10000000 Gwei). Value is set in Gwei format where 1,000,000,000 Gwei = 1 AVAX

```
const value: BN = new BN("10000000")
const amount: BN = value.sub(fee)
```

:::tip Snowtrace provides a [unit converter](#) between different units :::

Run the export script:

```
ts-node examples/avm/buildExportTx-cchain-avax.ts
```

This returns:

```
Success! TXID: 2uQvMcPZjmPXAyvz9cdKBphDDSmnxxx3vsUrxqpj3U92hsfQcc
```

#### Verify the Transaction

You can now pass this txID 2uQvMcPZjmPXAyvz9cdKBphDDSmnxxx3vsUrxqpj3U92hsfQcc into [examples/avm/getTx.ts](#), plus other similar network settings, then you can run

```
ts-node examples/avm/getTx.ts
```

which returns:

```
{
 unsignedTx: {
 networkID: 5,
 blockchainID: '2JVSBoinj9C2J33VntvzYtVJNZdN2NKiwwKjcumHUWEb5DbBrm',
 outputs: [[Object]],
 inputs: [[Object], [Object]],
 memo: '',
 destinationChain: 'yH8D7ThNJKxmtkuv2jgBa4P1Rn3Qpr4pPr7QYNfcdoS6k6HWp',
 exportedOutputs: [[Object]]
 },
 credentials: [
 {
 fxID: 'spdxUxVJQbX85MGxMHBKwlshxMnSqJ3QBzDyDYEP3h6TLuxqQ',
 credential: [Object]
 },
 {
 fxID: 'spdxUxVJQbX85MGxMHBKwlshxMnSqJ3QBzDyDYEP3h6TLuxqQ',
 credential: [Object]
 }
]
}
```

#### Import the Avax Token From X-Chain to C-Chain

Select the [examples/evm](#) folder to view the AvalancheJS C-Chain examples. To import AVAX to the C-Chain from the X-Chain, select [evm/buildImportTx-xchain.ts](#)

Copy the [network setting from above](#) into `evm/buildImportTx-xchain.ts`.

Navigate to this part of the code and ensure that the `cHexAddress` (Your C-Chain wallet address) and `private key` are correct:

```
const cHexAddress: string = "<YOUR-CCHAIN-WALLET-ADDRESS-HERE>"
const privKey: string = "<YOUR-PRIVATE-KEY-HERE>"
```

Run the import script:

```
ts-node examples/evm/buildImportTx-xchain.ts
```

This returns:

```
Success! TXID: 2uQvMcPZjmPXAvz9cdKBphDDSmnxx3vsUrxqpj3U92hsfQcc
```

That's it! You've transferred AVAX from the X-Chain to C-Chain!

You can verify this TX by copy / pasting the import TXID into [Avascan](#).

### Transfer from the C-Chain to the X-Chain

To return the AVAX back to the X-Chain, you need to do the transfer in the opposite direction.

#### Export the Avax Token From C-Chain to X-Chain

Select the [examples/evm](#) folder to view the AvalancheJS C-Chain examples. To export AVAX from the X-Chain to the C-Chain, select [evm/buildExportTx-xchain-avax.ts](#).

Make necessary changes as above for private key and network settings.

You can change the amount of AVAX to send by editing the *BN* variable: `avaxAmount`. The sample code assigns this as `1e7` or `10000000` (0.01 AVAX)

The fee here will only be for exporting the asset. The import fees will be deducted from the UTXOs present on the Exported Atomic Memory, a memory location where UTXOs stay after getting exported but before being imported.

```
let avaxAmount: BN = new BN(1e7)
let fee: BN = baseFee.div(new BN(1e9))
fee = fee.add(new BN(1e6))
```

Run the export script:

```
avalanchejs $ ts-node examples/evm/buildExportTx-xchain-avax.ts
Success! TXID: UAez3DTv26qmhKKFDvmQTayaXTPAVahHenDKe6xnUMhJbKuxc
```

#### Import the Avax Token From C-Chain to X-Chain

Before we run the [example import script](#), we need to make some changes to the code:

1. Change the [Network Setting](#) to meet Fuji network requirements.
2. Import your Private Key by following the steps listed [here](#).
3. Run the Script!

```
avalanchejs $ ts-node examples/avm/buildImportTx-cchain.ts
Success! TXID: Sm6Ec2GyguWyG3L1ipARmTpaz6qLEPuVAHV8QBGL9JWwWAEGM
```

## Mainnet Workflow

The Fuji workflow above can be adapted to Mainnet with the following modifications:

- The correct private key.
- Network setting should be to a Mainnet node, either [a local node on Mainnet](#) or [Avalanche Mainnet API server](#) where `api.avax.network` should be used for the `ip`.
- `const networkID: number = 1` based on [this](#).
- Set the correct amount to send.
- The correct receiving address.

## Local Workflow

### Start the Local Network

Follow [this](#) to start a 5-node local network. Make sure that you get one of the port number by following [this](#). In this tutorial, we will assume one of the ports is 30301.

### Locate the Example Code and Make Necessary Changes

Most of the code are already set to run it on a local network. Do check the following values to make sure they are correct.

```
const ip: string = "localhost"
const port: number = 30301 // Change this to the correct value
```

```
const protocol: string = "http"
const networkID: number = 1337
```

Then run the export and import scripts to transfer tokens across chains.

## Fuji Workflow

### Introduction

Fuji is the Avalanche network's test network. You can use it to test your dapp or smart contract after you've developed it locally. (You can use [Avalanche Network Runner](#) to test things locally.) Fuji is typically on the same version as the Avalanche Mainnet, but sometimes it is running an unreleased version of AvalancheGo. In general, you can expect Fuji's behavior to be about the same as Avalanche Mainnet. Tools such as a explorers and wallets should work with the Fuji Testnet.

In this tutorial, we'll go through an example Fuji workflow to show how it can be used. We'll do the following:

1. Set up Fuji network on MetaMask (optional)
2. Generate a 24 word english mnemonic via AvalancheJS
3. Derive external C-Chain addresses via AvalancheJS
4. Get AVAX from the Fuji faucet
5. Send AVAX via ethersJS
6. Examine the resulting transaction on the Avalanche Explorer
7. Use a private key derived from a mnemonic to sign into the web wallet

### Set up Fuji Network on MetaMask (optional)

- **Network Name:** Avalanche Fuji C-Chain
- **New RPC URL:** <https://api.avax-test.network/ext/bc/C/rpc>
- **ChainID:** 43113
- **Symbol:** AVAX
- **Explorer:** <https://testnet.snowtrace.io/>

### Generate a Mnemonic

To begin, we'll create a mnemonic phrase with [AvalancheJS](#). Mnemonics enable us to encode strong security into a human-readable phrase. AvalancheJS supports 10 languages including English, Japanese, Spanish, Italian, French, Korean, Czech, Portuguese, Chinese Simplified and Chinese Traditional.

First, generate a 24 word english [BIP39](#)-compliant mnemonic via AvalancheJS.

```
import { Mnemonic } from "avalanche"
const mnemonic: Mnemonic = Mnemonic.getInstance()
const strength: number = 256
const wordlist = mnemonic.getWordlists("english") as string[]
const m: string = mnemonic.generateMnemonic(strength, randomBytes, wordlist)
console.log(m)
// "chimney asset heavy ecology accuse window gold weekend annual oil emerge alley retreat rabbit seed advance define off amused board quick wealth peasant disorder"
```

### Derive Addresses

After generating a mnemonic we can use AvalancheJS to derive [BIP32](#)-compliant hierarchical deterministic (HD) Keypairs.

```
import HDNode from "avalanche/dist/utils/hdnode"
import { Avalanche, Mnemonic, Buffer } from "avalanche"
import { EVMAPI, KeyChain } from "avalanche/dist/apis/evm"
import { ethers } from "ethers"

const ip: string = "api.avax-test.network"
const port: number = 443
const protocol: string = "https"
const networkID: number = 5
const avalanche: Avalanche = new Avalanche(ip, port, protocol, networkID)
const cchain: EVMAPI = avalanche.CChain()

const mnemonic: Mnemonic = Mnemonic.getInstance()
const m: string =
 "chimney asset heavy ecology accuse window gold weekend annual oil emerge alley retreat rabbit seed advance define off amused
board quick wealth peasant disorder"
const seed: Buffer = mnemonic.mnemonicToSeedSync(m)
const hdnode: HDNode = new HDNode(seed)

const keyChain: KeyChain = cchain.newKeyChain()

const cAddresses: string[] = []

for (let i: number = 0; i <= 2; i++) {
 const child: HDNode = hdnode.derive(`m/44'/60'/0'/0/${i}`)
```

```

keyChain.importKey(child.privateKey)
const cchainAddress = ethers.utils.computeAddress(child.privateKey)
cAddresses.push(cchainAddress)
}
console.log(cAddresses)
// [
// '0x2d1d87fF3Ea2ba6E0576bCA4310fC057972F2559',
// '0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C',
// '0xa14dFb7d8593c44a47A07298eCEA774557036ff3'
//]

```

### Generate Private Keys from a Mnemonic

As long as you have the mnemonic phrase, you can re-generate your private keys and the addresses they control.

For example, if you want to generate the private keys for the first 3 address in the C Chain keychain:

- [0x2d1d87fF3Ea2ba6E0576bCA4310fC057972F2559](#)
- [0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C](#)
- [0xa14dFb7d8593c44a47A07298eCEA774557036ff3](#)

you might update the example script above to the following:

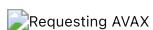
```

const cAddresses: string[] = []
const privateKeys: string[] = []
for (let i: number = 0; i <= 2; i++) {
 // Deriving the _i_th external BIP44 C-Chain address
 const child: HDNode = hdnode.derive(`m/44'/60'/0/${i}`)
 keyChain.importKey(child.privateKey)
 // Converting the BIP44 addresses to hexadecimal addresses
 const cchainAddress = ethers.utils.computeAddress(child.privateKey)
 privateKeys.push(child.privateKey.toString("hex"))
 cAddresses.push(cchainAddress)
}
console.log({ cAddresses, privateKeys })
// {
// cAddresses: [
// '0x2d1d87fF3Ea2ba6E0576bCA4310fC057972F2559',
// '0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C',
// '0xa14dFb7d8593c44a47A07298eCEA774557036ff3'
//],
// privateKeys: [
// 'cd30aef1af167238c627593537e162ecf5aad1d4ab4ea98ed2f96ad4e47006dc',
// 'b85479b26bc8fbada4737e90ab2133204f2fa2a9ea33c1e0de4452cbf8fa3be4',
// 'c72e18ea0f9aa5457396e3bf810e9de8df0177c8e4e5bf83a85f871512d645a9'
//]
// }

```

### Get a Drip from the Fuji Faucet

We can get a "drip" of AVAX from the Fuji faucet. Paste the address into the [Fuji faucet website](#). These AVAX are for the Fuji Testnet and have no monetary value.

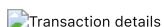


The faucet will send some AVAX to the address and return a transaction ID (txID). This txID can be used with the Fuji Testnet Explorer to learn more about the transaction.



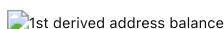
### Check the Transaction Details

The txID, `0x1419b04559bf140ab82216f7696110936fb7d4bc1f147e3b85fef7ca1008a19e`, can be seen on the [Fuji Testnet Explorer](#). Avalanche also has a [Mainnet Explorer](#).



### Get the Balance

We can also use the Fuji Explorer to get the balance for the 1st address—[0x2d1d87fF3Ea2ba6E0576bCA4310fC057972F2559](#).



Alternatively, we can use [ethersJS](#) to get the balance.

```

const ethers = require("ethers")
const network = "https://api.avax-test.network/ext/bc/C/rpc"
const provider = ethers.getDefaultProvider(network)
const address = "0x2d1d87fF3Ea2ba6E0576bCA4310fc057972F2559"

const main = async () : Promise<any> => {
 provider.getBalance(address).then((balance) => {
 // convert a currency unit from wei to ether
 const balanceInAvax = ethers.utils.formatEther(balance)
 console.log(`balance: ${balanceInAvax} AVAX`)
 // balance: 2 AVAX
 })
}

main()

```

## Sending AVAX

The faucet sent 2 AVAX to the first address we generated. Let's send AVAX from the 1st address to the 2nd address.

```

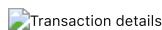
// import ethers.js
import { ethers } from "ethers"
// network: using the Fuji testnet
const network = "https://api.avax-test.network/ext/bc/C/rpc"
// provider: establish and RPC connection to the network
const provider = new ethers.providers.JsonRpcProvider(network)

// Sender private key:
// corresponding address 0x0x2d1d87fF3Ea2ba6E0576bCA4310fc057972F2559
let privateKey =
 "cd30ae1af167238c627593537e162ecf5aad1d4ab4ea98ed2f96ad4e47006dc"
// Create a wallet instance
let wallet = new ethers.Wallet(privateKey, provider)
// Receiver Address
let receiverAddress = "0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C"
// AVAX amount to send
let amountInAvax = "0.01"
// Create a transaction object
let tx = {
 to: receiverAddress,
 // Convert currency unit from ether to wei
 value: ethers.utils.parseEther(amountInAvax),
}
// Send a transaction
wallet.sendTransaction(tx).then((txObj) => {
 console.log(`tx, https://testnet.snowtrace.io/tx/${txObj.hash}`)
 // A transaction result can be checked in a snowtrace with a transaction link which can be obtained here.
})

```

## Verify Success

We can verify that the transaction, 0x3a5f4198b3be8d24b272f8255912aae4dcf2fb1f97f70d1787434de7b3097aac, was successful using the Fuji Testnet Explorer. The transaction can be seen [here](#).



## Get the Balance

We can also use the Fuji Explorer to get the balance for the 2nd address—[0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C](#).

Alternatively, we can use ethersJS to get the balance.

```

const ethers = require("ethers")
const network = "https://api.avax-test.network/ext/bc/C/rpc"
const provider = ethers.getDefaultProvider(network)
const address = "0x25d83F090D842c1b4645c1EFA46B15093d4CaC7C"

const main = async () : Promise<any> => {
 provider.getBalance(address).then((balance) => {
 // convert a currency unit from wei to ether
 const balanceInAvax = ethers.utils.formatEther(balance)
 console.log(`balance: ${balanceInAvax} AVAX`)
 // balance: 0.02 AVAX
 })
}

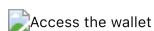
```

```
main()
```

## Sign Into the Web Wallet

Lastly, we can [use the mnemonic to generate a private key](#) to access the [Avalanche Web Wallet](#). We'll see that it has the AVAX balance and that it derives the hexadecimal address from the private key.

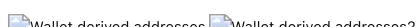
Use the private key to access the Web Wallet.



The balance is correct and the address is the 1st derived address.



We can repeat this login process using the private keys from the remaining 2 addresses in the [script above](#).



## Summary

The Fuji Testnet plays a critical role in testing dapps, smart contracts and financial products before deploying to the Mainnet. Tooling like AvalancheJS, the public API, faucet, and explorer helps to ensure that your testing and QA environment is close to Mainnet so that you can be confident when you launch on Mainnet.

## Resources

For additional and valuable resources please see below.

### Faucet

The [Fuji Faucet](#) sends AVAX to X-Chain or C-Chain addresses to help you test. (This testnet Avax has no value.)

### Wallet

The [Avalanche Web Wallet](#) is a simple, secure, non-custodial wallet for storing Avalanche assets. It supports Mainnet, Fuji and custom networks.

### Explorer

The Avalanche Explorer allows you to explore the network on [Mainnet](#) and [Fuji](#).

### Public API

See [here](#).

### AvalancheJS Examples

There are over [60 example AvalancheJS scripts](#) which demonstrate how to assets and NFTs, send transactions, add validators and more.

## C-Chain Exchange Integration

### Overview

The objective of this document is to provide a brief overview of how to integrate with the EVM-Compatible Avalanche C-Chain. For teams that already support ETH, supporting the C-Chain is as straightforward as spinning up an Avalanche node (which has the [same API](#) as [go-ethereum](#)) and populating Avalanche's ChainID (43114) when constructing transactions.

Additionally, Ava Labs maintains an implementation of the [Rosetta API](#) for the C-Chain called [avalanche-rosetta](#). You can learn more about this standardized integration path on the attached Rosetta API website.

### Integration Using EVM Endpoints

#### Running an Avalanche Node

If you want to build your node from source or include it in a docker image, reference the [AvalancheGo GitHub repository](#). To quickly get up and running, you can use the [node installation script](#) that automates installing and updating AvalancheGo node as a `systemd` service on Linux, using prebuilt binaries.

#### Configuring an Avalanche Node

All configuration options and their default values are described [here](#).

You can supply configuration options on the command line, or use a config file, which can be easier to work with when supplying many options. You can specify the config file location with `--config-file=config.json`, where `config.json` is a JSON file whose keys and values are option names and values.

Individual chains, including the C-Chain, have their own configuration options which are separate from the node-level options. These can also be specified in a config file. For more details, see [here](#).

The C-Chain config file should be at `$HOME/.avalanchego/configs/chains/C/config.json`. You can also tell AvalancheGo to look somewhere else for the C-Chain config file with option `--chain-config-dir`. An example C-Chain config file:

:::caution

If you need Ethereum [Archive Node](#) functionality, you need to disable C-Chain pruning, which has been enabled by default since AvalancheGo v1.4.10. To disable pruning, include `"pruning-enabled": false` in the C-Chain config file as shown below.

:::

```
{
 "snowman-api-enabled": false,
 "coreth-admin-api-enabled": false,
 "local-txs-enabled": true,
 "pruning-enabled": false,
 "eth-apis": [
 "internal-eth",
 "internal-blockchain",
 "internal-transaction",
 "internal-tx-pool",
 "internal-account",
 "internal-personal",
 "debug-tracer",
 "web3",
 "eth",
 "eth-filter",
 "admin",
 "net"
]
}
```

## Interacting with the C-Chain

Interacting with the C-Chain is identical to interacting with [go-ethereum](#). You can find the reference material for C-Chain API [here](#).

Please note that `personal_` namespace is turned off by default. To turn it on, you need to pass the appropriate command line switch to your node, like in the above config example.

## Integration Using Rosetta

[Rosetta](#) is an open-source specification and set of tools that makes integrating with different blockchain networks easier by presenting the same set of APIs for every network. The Rosetta API is made up of 2 core components, the [Data API](#) and the [Construction API](#). Together, these APIs allow for anyone to read and write to blockchains in a standard format over a standard communication protocol. The specifications for these APIs can be found in the [rosetta-specifications](#) repository.

You can find the Rosetta server implementation for Avalanche C-Chain [here](#), all you need to do is install and run the server with proper configuration. It comes with a `Dockerfile` that packages both the server and the Avalanche client. Detailed instructions can be found in the linked repository.

## Constructing Transactions

Avalanche C-Chain transactions are identical to standard EVM transactions with 2 exceptions:

- They must be signed with Avalanche's ChainID (43114).
- The detailed dynamic gas fee can be found [here](#).

For development purposes, Avalanche supports all the popular tooling for Ethereum, so developers familiar with Ethereum and Solidity can feel right at home. We have tutorials and repositories for several popular development environments:

- [MetaMask and Remix](#)
- [Truffle](#)
- [Hardhat](#)

## Ingesting On-Chain Data

You can use any standard way of ingesting on-chain data you use for Ethereum network.

### Determining Finality

Avalanche consensus provides fast and irreversible finality with 1-2 seconds. To query the most up-to-date finalized block, query any value (that is block, balance, state, etc) with the `latest` parameter. If you query above the last finalized block (that is `eth_blockNumber` returns 10 and you query 11), an error will be thrown indicating that unfinalized data cannot be queried (as of `avalanchego@v1.3.2`).

### (Optional) Custom Golang SDK

If you plan on extracting data from the C-Chain into your own systems using Golang, we recommend using our custom [ethclient](#). The standard [go-ethereum](#) Ethereum client does not compute block hashes correctly (when you call `block.Hash()`) because it doesn't take into account the added [ExtDataHash](#) header

field in Avalanche C-Chain blocks, which is used move AVAX between chains (X-Chain and P-Chain). You can read more about our multi-chain abstraction [here](#) (out of scope for a normal C-Chain integration).

If you plan on reading JSON responses directly or use web3.js (doesn't recompute hash received over the wire) to extract on-chain transaction data/logs/receipts, you shouldn't have any issues!

## Support

If you have any problems or questions, reach out either directly to our developers, or on our public [Discord](#) server.

# Multi Signature UTXOs with AvalancheJS

## Introduction

An account on a chain that follows the UTXO model doesn't have a parameter like balance. All it has is a bunch of outputs that are resulted from previous transactions. Each output has some amount of asset associated with them. These outputs can have 1 or multiple owners. The owners are basically the account addresses that can consume this output.

The outputs are the result of a transaction that can be spent by the owner of that output. For example, an account has 3 outputs that it can spend, and hence are currently unspent. That is why we call them Unspent Transaction Outputs (UTXOs). So it is better to use the term unspent outputs rather than just outputs. Similarly, we add the amount in the UTXOs owned by an address to calculate its balance. Signing a transaction basically adds the signature of the UTXO owners included in the inputs.

If an account A wants to send 1.3 AVAX to account B, then it has to include all those unspent outputs in a transaction, that are owned by A and whose sum of amounts in those outputs is more than or equal to 1.3. These UTXOs will be included as inputs in a transaction. Account A also has to create outputs with amount 1.3 and the owner being the receiver (here B). There could be multiple outputs in the outputs array. This means, that using these UTXOs, we can create multiple outputs with different amounts to different addresses.

Once the transaction is committed, the UTXOs in the inputs will be consumed and outputs will become new UTXOs for the receiver. If the inputs have more amount unlocked than being consumed by the outputs, then the excess amount will be burned as fees. Therefore, we should also create a change output which will be assigned to us, if there is an excess amount in the input. In the diagram given below, a total of 1.72 AVAX is getting unlocked in inputs, therefore we have also created a change output for the excess amount (0.41 AVAX) to the sender's address. The remaining amount after being consumed by the outputs like receiver's and change output, is burned as fees (0.01 AVAX).



## Multi-Signature UTXOs

UTXOs can be associated with multiple addresses. If there are multiple owners of a UTXO, then we must note the `threshold` value. We have to include signatures of a threshold number of UTXO owners with the unsigned transaction to consume UTXOs present in the inputs. The threshold value of a UTXO is set while issuing the transaction.

We can use these multi-sig UTXOs as inputs for multiple purposes and not only for sending assets. For example, we can use them to create Subnets, add delegators, add validators, etc.

## Atomic Transactions

On Avalanche, we can even create cross-chain outputs. This means that we can do a native cross-chain transfer of assets. These are made possible through **Atomic Transactions**. This is a 2-step process -

- Export transaction on source chain
- Import transactions on the destination chain

Atomic transactions are similar to other transactions. We use UTXOs of the source chain as inputs and create outputs owned by destination chain addresses. When the export transactions are issued, the newly created UTXOs stay in the **Exported Atomic Memory**. These are neither on the source chain nor on the destination chain. These UTXOs can only be used as inputs by their owners on the destination chain while making import transactions. Using these UTXOs on the atomic memory, we can create multiple outputs with different amounts or addresses.



## UTXOs on C-Chain

We can't use UTXOs on C-Chain to do regular transactions because C-Chain follows the account-based approach. In C-Chain, each address (account) is mapped with its balance, and the assets are transferred simply by adding and subtracting from this balance using the virtual machine.

But we can export UTXOs with one or multiple owners to C-Chain and then import them by signing the transaction with the qualified spenders containing those UTXOs as inputs. The output on C-Chain can only have a single owner (a hexadecimal address). Similarly while exporting from C-Chain to other chains, we can have multiple owners for the output, but input will be signed only by the account whose balance is getting used.

## Getting Hands-on Multi-Signature UTXOs

Next, we will make utility and other helpful functions, so that, we can use them to create multi-sig UTXOs and spend them with ease. These functions will extract common steps into a function so that we do not have to follow each step every time we are issuing a transaction.

You can either follow the steps below to get a better understanding of concepts and code or directly clone and test the examples from this [repo](#).

## Setting Up Project

Make a new directory `multisig` for keeping all the project codes and move there. First, let's install the required dependencies.

```
npm install --save avalanche dotenv
```

Now create a configuration file named `config.js` for storing all the pieces of information regarding the network and chain we are connecting to. Since we are making transactions on the Fuji network, its network ID is 5. You can change the configuration according to the network you are using.

```
require("dotenv").config()

module.exports = {
 protocol: "https",
 ip: "api.avax-test.network",
 port: 443,
 networkID: 5,
 privateKeys: JSON.parse(process.env.PRIVATEKEYS),
 mnemonic: process.env.MNEMONIC,
}
```

Create a `.env` file for storing sensitive information which we can't make public like the private keys or the mnemonic. Here are the sample private keys, which you should not use. You can create a new account on [Avalanche Wallet](#) and paste the mnemonic here for demonstration.

```
PRIVATEKEYS=[
 "PrivateKey-ewoqjP7PxY4yr3iLTpLisriqt94hdyDFNgchSxGGztUrTxtNN",
 "PrivateKey-R6e8f5QSa89DjpvL9asNdhdJ4u8VqzMJStPV8VVdDmLgPd8a4"
]
MNEMONIC="mask stand appear..."
```

## Setting Up APIs and Keychains

Create a file `importAPI.js` for importing and setting up all the necessary APIs, Keychains, addresses, etc. Now paste the following snippets into the file.

### Importing Dependencies and Configurations

We need dependencies like the AvalancheJS module and other configurations. Let's import them at the top.

```
const { Avalanche, BinTools, BN } = require("avalanche")
const Web3 = require("web3")

const MnemonicHelper = require("avalanche/dist/utils/mnemonic").default
const HDNode = require("avalanche/dist/utils/hdnode").default
const { privateToAddress } = require("ethereumjs-util")

// Importing node details and Private key from the config file.
const {
 ip,
 port,
 protocol,
 networkID,
 privateKeys,
 mnemonic,
} = require("./config.js")

let { avaxAssetID, chainIDs } = require("./constants.js")

// For encoding and decoding to CB58 and buffers.
const bintools = BinTools.getInstance()
```

### Setup Avalanche APIs

To make API calls to the Avalanche network and different blockchains like X-Chain, P-Chain and C-Chain, let's set up these by adding the following code snippet.

```
// Avalanche instance
const avalanche = new Avalanche(ip, port, protocol, networkID)
const nodeURL = `${protocol}://${ip}:${port}/ext/bc/C/rpc`
const web3 = new Web3(nodeURL)

// Platform and Avax API
const platform = avalanche.PChain()
const avax = avalanche.XChain()
const evm = avalanche.CChain()
```

### Setup Keychains with Private Keys

In order to sign transactions with our private keys, we will use the AvalancheJS keychain API. This will locally store our private keys and can be easily used for signing.

```
// Keychain for signing transactions
const keyChains = {
 x: avax.keyChain(),
 p: platform.keyChain(),
 c: evm.keyChain(),
}

function importPrivateKeys(privKey) {
 keyChains.x.importKey(privKey)
 keyChains.p.importKey(privKey)
 keyChains.c.importKey(privKey)
}
```

We can either use mnemonics to derive private keys from it or simply use the bare private key for importing keys to the keychain. We can use the following function to get private keys from the mnemonic and address index which we want. For demo purposes, we will use addresses at index 0 and 1.

```
function getPrivateKey(mnemonic, activeIndex = 0) {
 const mnemonicHelper = new MnemonicHelper()
 const seed = mnemonicHelper.mnemonicToSeedSync(mnemonic)
 const hdNode = new HDNode(seed)

 const avaPath = `m/44'/9000'/0'/0/${activeIndex}`

 return hdNode.derive(avaPath).privateKeyCB58
}

// importing keys in the key chain - use this if you have any private keys
// privateKeys.forEach((privKey) => {
// importPrivateKeys(privKey)
// })

// importing private keys from mnemonic
importPrivateKeys(getPrivateKey(mnemonic, 0))
importPrivateKeys(getPrivateKey(mnemonic, 1))
```

### Setup Addresses and Chain IDs

For creating transactions we might need addresses of different formats like `Buffer` or `Bech32` etc. And to make issue transactions on different chains we need their `chainID`. Paste the following snippet to achieve the same.

```
// Buffer representation of addresses
const addresses = {
 x: keyChains.x.getAddresses(),
 p: keyChains.p.getAddresses(),
 c: keyChains.c.getAddresses(),
}

// String representation of addresses
const addressStrings = {
 x: keyChains.x.getAddressStrings(),
 p: keyChains.p.getAddressStrings(),
 c: keyChains.c.getAddressStrings(),
}

avaxAssetID = bintools.cb58Decode(avaxAssetID)

chainIDs = {
 x: bintools.cb58Decode(chainIDs.x),
 p: bintools.cb58Decode(chainIDs.p),
 c: bintools.cb58Decode(chainIDs.c),
}

// Exporting these for other files to use
module.exports = {
 networkID,
 platform,
 avax,
 evm,
 keyChains,
 avaxAssetID,
 addresses,
 addressStrings,
 chainIDs,
```

```
bintools,
web3,
BN,
}
```

We can use the above-exported variables and APIs from other files as required.

## Creating Utility Functions

While creating multi-sig transactions, we have a few things in common, like creating inputs with the UTXOs, creating outputs, and adding signature indexes. So let's create a file named `utils.js` and paste the following snippets that we can call every time we want to do a repetitive task.

### Getting Dependencies

Inputs and outputs are an array of transferable input and transferable output. These contain transfer inputs and associated assetID which is being transferred. There are different types of transfer inputs/outputs for sending assets, minting assets, minting NFTs, etc.

We will be using `SECPTTransferInput/SECPTTransferOutput` for sending our assets.

But since we can't use UTXOs on C-Chain, we cannot directly import them either. Therefore we need to create a different type of input/output for them called `EVMInput/EVMOutput`.

```
const { BN, chainIDs, web3 } = require("./importAPI")

let SECPTTransferInput,
TransferableInput,
SECPTTransferOutput,
TransferableOutput,
EVMInput,
EVMOutput

const getTransferClass = (chainID) => {
 let vm = ""
 if (chainID.compare(chainIDs.x) == 0) {
 vm = "avm"
 } else if (chainID.compare(chainIDs.p) == 0) {
 vm = "platformvm"
 } else if (chainID.compare(chainIDs.c) == 0) {
 vm = "evm"
 }
 return {{
 SECPTTransferInput,
 TransferableInput,
 SECPTTransferOutput,
 TransferableOutput,
 EVMInput,
 EVMOutput,
 index,
 } = require(`avalanche/dist/apis/${vm}/index`)
}
```

Different chains have their own implementation of TransferInput/Output classes. Therefore we need to update the required modules according to the chain we issuing transactions on. To make it more modular, we created a `getTransferClass()` function, that will take `chainID` and import modules as required.

### Creating Transferable Output

The `createOutput()` function will create and return the transferable output according to arguments amount, assetID, owner addresses, lock time, and threshold. Lock time represents the timestamp after which this output could be spent. Mostly this parameter will be 0.

```
const createOutput = (amount, assetID, addresses, locktime, threshold) => {
 let transferOutput = new SECPTTransferOutput(
 amount,
 addresses,
 locktime,
 threshold
)

 return new TransferableOutput(assetID, transferOutput)
}
```

### Creating Transferable Input

The `createInput()` function will create and return transferable input. Input require arguments like amount in the UTXO, and arguments which identify that UTXO, like txID of the transaction which the UTXO was the output of, `outputIndex` (index of the output in that TX), and qualified signatures (output spenders which are present in our keychain) whose signature will be required while signing this transaction.

```

const createInput = (
 amount,
 txID,
 outputIndex,
 assetID,
 spenders,
 threshold
) => {
 // creating transfer input
 let transferInput = new SECPTTransferInput(amount)

 // adding threshold signatures
 addSignatureIndexes(spenders, threshold, transferInput)

 // creating transferable input
 return new TransferableInput(txID, outputIndex, assetID, transferInput)
}

```

### Add Signature Indexes

The `createSignatureIndexes()` function will add spender addresses along with an index for each address in the transfer input. While signing the unsigned transaction, these signature indexes will be used.

By adding signature indexes we are not signing the inputs but just adding a placeholder of the address at a particular index whose signature is required when we call the `.sign()` function on the unsigned transactions. Once the threshold spender addresses are added, it will exit.

```

const addSignatureIndexes = (addresses, threshold, input) => {
 let sigIndex = 0
 addresses.every((address) => {
 if (threshold > 0) {
 input.addSignatureIdx(sigIndex, address)
 sigIndex++
 threshold--
 return true
 } else {
 return false
 }
 })
}

```

### Create EVM Input

As explained earlier, we do not have UTXOs on C-Chain. Therefore we cannot make regular inputs. The following function `createEVMInput()` will create the required input and add a signature index corresponding to the address specified in the input.

EVM Inputs are required when we want to export assets from C-Chain. In the following function, `addresses` is the array of Buffer addresses but for `C-Chain Export Transactions`, a hex address is also appended at last.

```

const createEVMInput = (amount, addresses, assetID, nonce) => {
 const hexAddress = addresses.at(-1)
 const evmInput = new EVMInput(hexAddress, amount, assetID, nonce)
 evmInput.addSignatureIdx(0, addresses[0])

 return evmInput
}

```

### Create EVM Output

The `createEVMOutput()` function will create EVM output for importing assets on C-Chain.

```

const createEVMOutput = (amount, hexAddress, assetID) => {
 return new EVMOutput(hexAddress, amount, assetID)
}

```

### Update Transfer Class

Let's make a small function that will call the `getTransferClass()` according to the `chainID`.

```

const updateTransferClass = (chainID) => {
 {
 SECPTTransferInput,
 TransferableInput,
 SECPTTransferOutput,
 TransferableOutput,
 EVMInput,
 }
}

```

```

 EVMOutput,
 (index = getTransferClass(chainID)
}
}

```

### Add UTXOs to Inputs

We have `inputs` as an array of UTXOs that will be consumed in the transaction. The `updateInputs()` function will take UTXOs, `addresses` whose credentials we have for signing, `assetID` and `toBeUnlocked` that is amount we want to consume. `toBeUnlocked` contains everything we want to consume including transfer amount, fees, stake amount (if any), etc.

We also have a special variable `C`, that will indicate the type of transaction which is associated with the C-Chain. This is required because -

- Export from C-Chain (`C.export == true`) - These types of transactions cannot have UTXOs as inputs and therefore `EVMInput` is created.
- Import to C-Chain (`C.import == true`) - The outputs imported on C-Chain from exported UTXOs are `EVMOutput`.

It will create inputs with the passed UTXOs worth the `toBeUnlocked` amount. But if there is a UTXO that when included, will surpass the `toBeUnlocked` amount, then it will create a change output with the qualified spenders as their new owners with the surpassed amount.

This function will return the `inputs` array containing all the unlocked UTXOs, change transferable output, and the net balance included in these inputs. Now add the following function snippet.

```

const updateInputs = async (
 utxos,
 addresses,
 C,
 assetID,
 toBeUnlocked,
 chainID
) => {
 // Getting transferable inputs according to chain id
 updateTransferClass(chainID)

 let inputs = [],
 changeTransferableOutput = undefined,
 netInputBalance = new BN(0)

 if (C.export) {
 const nonce = await web3.eth.getTransactionCount(addresses.at(-1))
 inputs.push(createEVMInput(toBeUnlocked, addresses, assetID, nonce))
 } else {
 utxos.forEach((utxo) => {
 let output = utxo.getOutput()
 if (
 output.getOutputID() === 7 &&
 assetID.compare(utxo.getAssetID()) === 0 &&
 netInputBalance < toBeUnlocked
) {
 let outputThreshold = output.getThreshold()

 // spenders which we have in our keychain
 let qualifiedSpenders = output.getSpenders(addresses)

 // create inputs only if we have custody of threshold or more number of utxo spenders
 if (outputThreshold <= qualifiedSpenders.length) {
 let txID = utxo.getTxID()
 let outputIndex = utxo.getOutputIdx()
 let utxoAmount = output.amountValue
 let outputLocktime = output.getLocktime()

 netInputBalance = netInputBalance.add(utxoAmount)

 let excessAmount = netInputBalance.sub(toBeUnlocked)

 // creating change transferable output
 if (excessAmount > 0) {
 if (!C.import) {
 changeTransferableOutput = createOutput(
 excessAmount,
 assetID,
 qualifiedSpenders,
 outputLocktime,
 outputThreshold
)
 }
 }
 }
 }
 })
 }
}

// create transferable input

```

```

 let transferableInput = createInput(
 utxoAmount,
 txID,
 outputIndex,
 assetID,
 qualifiedSpenders,
 outputThreshold
)

 inputs.push(transferableInput)
 }
 })
 })

 return { inputs, changeTransferableOutput }
}

```

Only those UTXOs will be included whose output ID is `7` representing `SECPTransferOutput`. These outputs are used for transferring assets. Also, we are only including outputs containing `AVAX` assets. These conditions are checked in the following line -

```
if(output.getOutputID() === 7 && assetID.compare(utxo.getAssetID()) === 0 && netInputBalance < toBeUnlocked) {
```

The following part in the above function creates the change output if the total included balance surpasses the required amount and the transaction is not a C-Chain export -

```

netInputBalance = netInputBalance.add(utxoAmount)

let excessAmount = netInputBalance.sub(toBeUnlocked)

// creating change transferable output
if (excessAmount > 0) {
 if (!C.import) {
 changeTransferableOutput = createOutput(
 excessAmount,
 assetID,
 qualifiedSpenders,
 outputLocktime,
 outputThreshold
)
 }
}

```

## Export Utility Functions

Now paste the following snippet to export these utility functions.

```
module.exports = {
 createOutput,
 createEVMOutput,
 updateInputs,
}
```

All the utility functions are created.

## Create Inputs and Outputs

Let's create a function that will return the array of sufficient UTXOs stuffed inside an array and necessary outputs like send output, multi-sig output, evm output, change output, etc. This function is basically a wrapper that orchestrates the utility and other functions to generate inputs and outputs from parameters like addresses, asset id, chain id, output arguments (to, threshold and amount), etc.

Now make a new file `createInputsAndOutputs.js` and paste the following snippets of code inside it.

### Importing Dependencies

We need to import utility functions for creating outputs and inputs with the UTXOs.

```
const { BN, avax, platform, evm, chainIDs, bintools } = require("./importAPI")

const { createOutput, createEVMOutput, updateInputs } = require("./utils")
```

`EVMInput` should be used as inputs while creating an export transaction from C-Chain and `EVMOutput` should be used as outputs while creating an import transaction on C-Chain. To make it easier to decide when to do what, let's make a function `checkChain()` that will return an object `C` (described earlier).

```
const checkChain = (chainID, ownerAddress) => {
 let C = {
```

```

 export: false,
 import: false,
 }
 if (chainID.compare(chainIDs.c) == 0) {
 if (typeof ownerAddress == "string" && bintools.isHex(ownerAddress)) {
 C.import = true
 } else {
 C.export = true
 }
 }
 return C
}

```

For getting UTXOs from an address, let's make another function `getUnspentOutputs()`. This function will fetch UTXOs from a given address and source chain. The `sourceChain` will be used to fetch exported UTXOs that are not yet imported. The exported outputs stay in the exported atomic memory. This parameter will only be used when we want to import assets.

```

// UTXOs for spending unspent outputs
const getUnspentOutputs = async (
 addresses,
 chainID,
 sourceChain = undefined
) => {
 let utxoSet
 if (chainID.compare(chainIDs.x) == 0) {
 utxoSet = await avax.getUTXOs(addresses, sourceChain)
 } else if (chainID.compare(chainIDs.p) == 0) {
 utxoSet = await platform.getUTXOs(addresses, sourceChain)
 }
 return utxoSet.utxos.getAllUTXOs()
}

```

Now for organizing inputs and outputs and adding required signature indexes (not signatures) for each unspent output, adding change output, etc, we will make a `createInputsAndOutputs()` function. Paste the following snippet next.

```

const createInputsAndOutputs = async (
 assetID,
 chainID,
 addresses,
 addressStrings,
 outputConfig,
 fee,
 sourceChain
) => {
 let locktime = new BN(0)

 let C = checkChain(chainID, outputConfig[0].owners)

 let utxos = []
 if (C.export) {
 addresses.push("0x3b0e59fc2e9a82fa5eb3f042bc5151298e4f2cab") // getHexAddress(addresses[0])
 } else {
 utxos = await getUnspentOutputs(addressStrings, chainID, sourceChain)
 }

 let toBeUnlocked = fee
 outputConfig.forEach((output) => {
 toBeUnlocked = toBeUnlocked.add(output.amount)
 })

 // putting right utxos in the inputs
 let { inputs, changeTransferableOutput } = await updateInputs(
 utxos,
 addresses,
 C,
 assetID,
 toBeUnlocked,
 chainID
)

 let outputs = []

 // creating transferable outputs and transfer outputs
 outputConfig.forEach((output) => {
 let newOutput
 if (!C.import) {

```

```

 newOutput = createOutput(
 output.amount,
 assetID,
 output.owners,
 locktime,
 output.threshold
)
 } else {
 newOutput = createEVMOuput(output.amount, output.owners, assetID)
 }
 outputs.push(newOutput)
})

// pushing change output (if any)
if (changeTransferableOutput != undefined && !C.import) {
 outputs.push(changeTransferableOutput)
}

return { inputs, outputs }
}

```

Output config is basically an array of all outputs that we want to create. This excludes the change output because it will be automatically created. It has the following structure.

```

// Regular outputs
>[
{
 amount: BigNumber,
 owners: [Buffer],
 threshold: Number,
},
] [
 // Import to C-Chain
{
 amount: BigNumber,
 owners: "hex address string",
}
]

```

You will learn about these arguments and how we can actually pass this along with other arguments through the examples ahead.

## Exporting Functions

Add the following snippet to export this function.

```

module.exports = {
 createInputsAndOutputs,
}

```

We have created all the utility and helper functions. You can use this project structure to create different types of transactions like BaseTx, Export, Import, AddDelegator, etc. You should have the following files in your project now -

- **.env** - Secret file storing data like mnemonic and private keys
- **config.js** - Network information and parsed data from **.env**
- **constants.js** - Asset and Chain specific static data
- **importAPI.js** - Import and setup apis, addresses and keychains
- **utils.js** - Utility functions for creating inputs and outputs
- **createInputsAndOutputs.js** - Wrapper of **utility.js** for orchestrating utility functions.

Follow the next steps for **examples** and on how to use these functions.

## Examples

Now let's look at the examples for executing these transactions. For example, we will create a separate `examples` folder. In order to run the example scripts, you must be in the root folder where all the environment variables and configurations are kept.

```
node examples/send.js
```

## Multi-Signature Base TX on X-Chain

Let's create a base transaction that converts a single-owner UTXO into a multi-sig UTXO. The final UTXO can be used by new owners of the unspent output by adding their signatures for each output. Create a new file `sendBaseTx.js` and paste the following snippets.

### Import Dependencies

Import the necessary dependencies like `keyChains`, `addresses`, `utility` functions, `UnSignedTx` and `BaseTx` classes etc.

```

const {
 avaxAssetID,
 keyChains,
 chainIDs,
 addresses,
 addressStrings,
 networkID,
 BN,
 avax,
} = require("../importAPI")

const { UnsignedTx, BaseTx } = require("avalanche/dist/apis/avm/index")

const { createInputsAndOutputs } = require("../createMultisig")

```

### Send BaseTx

Now create the `sendBaseTx()` function to be called for sending base TX to the network.

```

async function sendBaseTx() {
 let memo = Buffer.from("Multisig Base Tx")

 // unlock amount = sum(output amounts) + fee
 let fee = new BN(1e6)

 // creating outputs of 0.5 (multisig) and 0.1 AVAX - change output will be added by the function in the last
 let outputConfig = [
 {
 amount: new BN(5e8),
 owners: addresses.x,
 threshold: 2,
 },
 {
 amount: new BN(1e8),
 owners: [addresses.x[1]],
 threshold: 1,
 },
]
}

let { inputs, outputs } = await createInputsAndOutputs(
 avaxAssetID,
 chainIDs.x,
 addresses.x,
 addressStrings.x,
 outputConfig,
 fee
)

const baseTx = new BaseTx(networkID, chainIDs.x, outputs, inputs, memo)

const unsignedTx = new UnsignedTx(baseTx)
const tx = unsignedTx.sign(keyChains.x)
const txID = await avax.issueTx(tx)
console.log("TxID:", txID)
}

```

We have created the BaseTx with the following output configuration -

- Multi-sig output of value 0.5 AVAX with threshold 2 and owners represented by `addresses.x`. The owners are basically an array of addresses in Buffer representation.
- Single owner output of value 0.1 AVAX.

```

let outputConfig = [
 {
 amount: new BN(5e8),
 owners: addresses.x,
 threshold: 2,
 },
 {
 amount: new BN(1e8),
 owners: [addresses.x[1]],
 threshold: 1,
 },
]

```

Let's discuss the arguments of `createInputsAndOutputs()` in detail -

- `assetID` - ID of the asset involved in transaction
- `chainID` - ID of the chain on which this transaction will be issued
- `addresses` - Addresses buffer array whose UTXO will be consumed
- `addressStrings` - Addresses string array whose UTXO will be consumed
- `outputConfig` - Array of output object containing amount, owners and threshold
- `fee` - Fee for this transaction to be consumed in inputs
- `sourceChain` - Chain from which UTXOs will be fetched. Will take `chainID` as default.

In the above parameters, if `fee` is less than the fees actually required for that transaction, then there will be no surplus amount left by outputs over inputs because any surplus will be converted into a change output. This can cause transaction failure. So keep the fees in accordance with the transaction as mentioned [here](#).

Also, the `sourceChain` parameter is required for fetching exported UTXOs that do not exist yet on the destination chain. For non-export/import transactions, this parameter is not required.

The `createInputsAndOutputs()` function will return `inputs` and `outputs` required for any transaction. The last element of the `outputs` array would be change output. And the order of other outputs will be the same as that in the `outputConfig`. Signature indexes corresponding to their owners are already included in the inputs. We can create an unsigned base transaction using the `BaseTx` and `UnsignedTx` classes as shown above. The `.sign()` function basically adds the required signatures from the keychain at the place indicated by signature indexes.

Once the multi-sig UTXO is created, this UTXO can only be used if we have the threshold signers in our keychain. The `util` functions can be tweaked a little bit to create and return inputs with a part number of signers (`<threshold`). We can then partially sign the inputs and ask other owners to add signature index and sign.

Now call the `sendBaseTx()` function by adding this line

```
sendBaseTx()
```

Run this file using `node examples/sendBaseTx.js`, see the txID in the output, and look for it in the Fuji explorer.



## Export Multi-Sig UTXO From X to P-Chain

Now we will look into exporting assets from the X to P chain. It will be similar to the `BaseTx` example, with few differences in output ordering and cross-chain owner addresses.

Make a new file named `exportXP.js` and paste the following snippets.

### Import Dependencies

This time we will require `ExportTx` instead of `BaseTx` class.

```
const {
 avaxAssetID,
 keyChains,
 chainIDs,
 addresses,
 addressStrings,
 networkID,
 BN,
 avax,
} = require("../importAPI")

const { UnsignedTx, ExportTx } = require("avalanche/dist/apis/avm/index")

const { createInputsAndOutputs } = require("../createMultisig")
```

### Send Export Transaction

Most of the things will be very much similar in this function. You can have a look at `outputConfig`, which creates a multi-sig output for addresses on P-Chain. These addresses will be required for signing `importTx` on P-Chain.

The `fee` here will only be for exporting the asset. The import fees will be deducted from the UTXOs present on the **Exported Atomic Memory**, a memory location where UTXOs lie after getting exported but before being imported. If there is only a single UTXO, then it will be deducted from it.

```
async function exportXP() {
 let memo = Buffer.from("Multisig Export Tx")

 // consuming amount = sum(output amount) + fee
 let fee = new BN(1e6)

 // creates multi-sig (0.1 AVAX) and single-sig (0.03 AVAX) output for exporting to P Address (0.001 AVAX will be fees)
 let outputConfig = [
 {
 amount: new BN(3e6),
 owners: [addresses.p[0]],
 }
]
}
```

```

 threshold: 1,
 },
 {
 amount: new BN(1e8),
 owners: addresses.p,
 threshold: 2,
 },
]

// importing fees will be deducted from these our other outputs in the exported output memory
let { inputs, outputs } = await createInputsAndOutputs(
 avaxAssetID,
 chainIDs.x,
 addresses.x,
 addressStrings.x,
 outputConfig,
 fee
)

// outputs at index 0 and 1 are to be exported
const exportTx = new ExportTx(
 networkID,
 chainIDs.x,
 [outputs.at(-1)],
 inputs,
 memo,
 chainIDs.p,
 [outputs[0], outputs[1]]
)

const unsignedTx = new UnsignedTx(exportTx)
const tx = unsignedTx.sign(keyChains.x)
const txID = await avax.issueTx(tx)
console.log("TxID:", txID)
}

```

Another point to note is how `inputs`, `outputs`, and `exportedOutputs` are passed here.

- Inputs are as usual passed for the `ins` parameter of the `ExportTx` class.
- But only `outputs`. `at(-1)` representing change output (last element) is passed in place of the usual `outs` parameter.
- The last parameter of this class is `exportedOuts`, representing the outputs that will be exported from this chain to `destinationChain` (2nd last parameter).

All these inputs and outputs are array, and hence `con` contains multiple outputs or inputs. But you have to manage which output should be passed where.

Call the function by adding the below function call.

```
exportXP()
```

Run this file using `node examples/exportXP.js`, see the `txID` in the output, and look for it in the [Fuji explorer](#).



In the above image, we are consuming UTXO with the amount `0.486...`, and generating outputs with the amount `0.382...` (change output) and `0.003` and `0.1` (exported output). The remaining `0.001` is burned as transaction fees.

## Import Multi-Sig UTXO From X to P-Chain

After exporting the UTXOs from the source chain, it stays in the exported atomic memory that is these are neither on the source chain nor on the destination chain. Paste the following snippets into a new file `importP.js`.

### Import Dependencies

We will require `ImportTx` from PlatformVM APIs.

```

const {
 avaxAssetID,
 keyChains,
 chainIDs,
 addresses,
 addressStrings,
 networkID,
 BN,
 platform,
} = require("../importAPI")

const { UnsignedTx, ImportTx } = require("avalanche/dist/apis/platformvm/index")

```

```
const { createInputsAndOutputs } = require("../createMultisig")
```

### Send Import Transaction

The `importP()` is a simple function that will use UTXOs on the exported atomic memory as its inputs and create an output on the P-Chain addresses. You can change the output config's owners and amount as per your need.

An important point to note here is that all UTXOs that are included in this `importTx` will be transferred to the destination chain. Even if the import amount is less than the amount in the UTXO, it will be sent to the qualified spender on the destination chain as a change output.

```
async function importP() {
 let memo = Buffer.from("Multisig Import Tx")

 // Use this parameter if you have UTXOs exported from other chains - only exported outputs will be fetched
 let sourceChain = "X"

 // unlock amount = sum(output amount) + fee
 let fee = new BN(1e6)

 let outputConfig = [
 {
 amount: new BN(1e6),
 owners: addresses.p,
 threshold: 2,
 },
 {
 amount: new BN(1e2),
 owners: addresses.p[0],
 threshold: 1,
 },
],
]

 // all the inputs here are the exported ones due to source chain parameter
 let { inputs, outputs } = await createInputsAndOutputs(
 avaxAssetID,
 chainIDs.p,
 addresses.p,
 addressStrings.p,
 outputConfig,
 fee,
 sourceChain
)

 const importTx = new ImportTx(
 networkID,
 chainIDs.p,
 outputs,
 [],
 memo,
 chainIDs.x,
 inputs
)

 const unsignedTx = new UnsignedTx(importTx)
 const tx = unsignedTx.sign(keyChains.x)
 const txID = await platform.issueTx(tx)
 console.log("TxID:", txID)
}
```



In the above image, we are consuming the above exported UTXOs with amounts `0.003` and `0.1`, and generating outputs with amount `0.092...` (change output imported on P-Chain) and 2 `0.005` imported outputs (1 multi-sig and 1 single-sig). The remaining `0.001` is burned as transaction fees.

### Import Multi-Sig UTXO From X to C-Chain

This transaction will also be similar to other atomic transactions, except for the `outputConfig` parameter. You can easily get the idea by looking at the code below. Before you can run this example, there must be exported outputs for the addresses you control on the C-Chain, otherwise, there will be no UTXO to consume.

Here we are importing UTXOs that are exported from X-Chain.

```
const {
 avaxAssetID,
 keyChains,
```

```

chainIDs,
addresses,
addressStrings,
networkID,
BN,
evm,
} = require("../importAPI")

const { UnsignedTx, ImportTx } = require("avalanche/dist/apis/evm/index")

const { createInputsAndOutputs } = require("../createMultisig")

async function importP() {
 // Use this parameter if you have UTXOs exported from other chains - only exported outputs will be fetched
 let sourceChain = "X"

 // unlock amount = sum(output amount) + fee (fees on C-Chain is dynamic)
 let fee = new BN(0)

 let outputConfig = [
 {
 amount: new BN(1e4),
 owners: "0x4406a53c35D05424966bd8FC354E05a3c6B56aF0",
 },
 {
 amount: new BN(2e4),
 owners: "0x3b0e59fc2e9a82fa5eb3f042bc5151298e4f2cab",
 },
]

 // all the inputs here are the exported ones due to source chain parameter
 let { inputs, outputs } = await createInputsAndOutputs(
 avaxAssetID,
 chainIDs.c,
 addresses.c,
 addressStrings.c,
 outputConfig,
 fee,
 sourceChain
)

 const importTx = new ImportTx(
 networkID,
 chainIDs.c,
 chainIDs.x,
 inputs,
 outputs
)

 const unsignedTx = new UnsignedTx(importTx)
 const tx = unsignedTx.sign(keyChains.x)
 const txID = await evm.issueTx(tx)
 console.log("TxID:", txID)
}

importP()

```



You can use [Avascan](#) to view import and export transactions on C-Chain.

## Add Delegator Transaction

Till now we have covered common transactions like BaseTx, Export, and Import TX. Export and Import TX will be similar in all the UTXO-based chains like X and P. But for Account-based chains, we have to deal with an account-balance system.

Now let's try using the multi-sig UTXOs exported from X-Chain to P-Chain to issue an `addDelegator()` transaction. Create a file `addDelegatorTx.js` and paste the following snippets.

### Import Dependencies

Import the dependencies like `AddDelegatorTx` and `UnsignedTx` classes using the following code.

```

const {
 avaxAssetID,
 keyChains,
 chainIDs,

```

```

addresses,
addressStrings,
networkID,
BN,
platform,
} = require("../importAPI")

const {
 UnsignedTx,
 AddDelegatorTx,
 SECPowerOutput,
 ParseableOutput,
} = require("avalanche/dist/apis/platformvm/index")

const { NodeIDStringToBuffer, UnixNow } = require("avalanche/dist/utils")

const { createInputsAndOutputs } = require("../createMultisig")

```

### Sending AddDelegator Transaction

Now we will create the `addDelegator()` function which will use the multi-sig UTXOs and create a signed `addDelegatorTx`, which when issued, will add the delegator to the specified node. Paste the following snippet next.

```

async function addDelegator() {
 let nodeID = NodeIDStringToBuffer("NodeID-4B4rc5vdD1758JSBYL1xyvE5NHGzz6xzH")
 let locktime = new BN(0)
 let stakeAmount = await platform.getMinStake()
 let startTime = UnixNow().add(new BN(60 * 1))
 let endTime = startTime.add(new BN(2630000))
 let memo = Buffer.from("Multi-sig Add Delegator Tx")

 // unlock amount = sum(output amounts) + fee
 let fee = new BN(1e6)

 // creating stake amount output at 0th index
 let outputConfig = [
 {
 amount: stakeAmount.minValidatorStake,
 owners: addresses.p,
 threshold: 2,
 },
]

 // outputs to be created for rewards
 const rewardOutputOwners = new SECPowerOutput(addresses.p, locktime, 2)
 const rewardOwners = new ParseableOutput(rewardOutputOwners)

 let { inputs, outputs } = await createInputsAndOutputs(
 avaxAssetID,
 chainIDs.p,
 addresses.p,
 addressStrings.p,
 outputConfig,
 fee
)

 const addDelegatorTx = new AddDelegatorTx(
 networkID,
 chainIDs.p,
 [],
 inputs,
 memo,
 nodeID,
 startTime,
 endTime,
 stakeAmount.minDelegatorStake,
 [outputs[0]],
 rewardOwners
)

 const unsignedTx = new UnsignedTx(addDelegatorTx)
 const tx = unsignedTx.sign(keyChains.p)
 const txID = await platform.issueTx(tx)
 console.log("TxID:", txID)
}

```

In the above transaction, the `outputs` parameter will be empty since we do not need to transfer any assets to the account. As you can see above we need to create another type of output, for indicating the reward for delegation.

```
const rewardOutputOwners = new SECPOwnerOutput(addresses.p, locktime, 2)
const rewardOwners = new ParseableOutput(rewardOutputOwners)
```

Call the function by adding the below function call.

```
addDelegator()
```

Run this file using `node examples/addDelegatorTx.js`, see the txID in the output, and look for it in the Fuji explorer.



**description:** This tutorial will help users to send transactions with dynamic fee settings to adjust their priority fee and max fee cap during high network activity using javascript.

## Sending Transactions with Dynamic Fees using JavaScript

### Overview

The objective of this document is to provide and explain sending transactions with dynamic fees using JavaScript. Make sure you have followed [the tutorial on adjusting the dynamic fees using MetaMask](#). There, we have explained the key concepts related to dynamic fees and EIP1559 type of transactions.

### Prerequisites

- Basic familiarity with [JavaScript](#).
- Basic familiarity with [Node.js](#) and [npm](#).
- Basic familiarity with the [Avalanche C-Chain](#) network and [EVM compatibility](#).
- Basic understanding of [dynamic fee transactions](#) transactions

### Installing Dependencies

Open the terminal and install the following dependencies in a new folder.

- Ethers
- avalanche
- dotenv

```
npm install ethers avalanche dotenv
```

### Setting up Environment and Project

To send a transaction we need to sign it using our private key. But private key should not be hard coded in the code, rather must be fetched through some environment variables. Make a `.env` file in the root folder with the following content.

```
PRIVATEKEY=<YOUR_PRIVATE_KEY>
```

Now make a new file `app.js` in the root folder, which will be our main and only file with the `sendAvax()` function. Follow the rest of the tutorial by understanding and pasting the provided snippets sequentially in the `app.js` file.

### Importing Dependencies and Private Key

```
const ethers = require("ethers")
const Avalanche = require("avalanche").Avalanche
require("dotenv").config()

const privateKey = process.env.PRIVATEKEY
```

### Setting up HTTP Provider Connected with Fuji Network

Using the HTTP provider, we will connect to one of the nodes on the Fuji network. Using this provider we will send the signed transaction to the network. You can also connect to Mainnet using the URL - <https://api.avax.network/ext/bc/C/rpc>

```
// For sending a signed transaction to the network
const nodeURL = "https://api.avax-test.network/ext/bc/C/rpc"
const HTTPSPProvider = new ethers.providers.JsonRpcProvider(nodeURL)
```

## Setting up C-Chain APIs for Estimating Base and Priority Fees

To estimate the max fee and max priority fee on the network, we will be using C-Chain APIs. We can use the C-Chain through an AvalancheJS instance connected to the network as shown below.

```
// For estimating max fee and priority fee using CChain APIs
const chainId = 43113
const avalanche = new Avalanche(
 "api.avax-test.network",
 undefined,
 "https",
 chainId
)
const cchain = avalanche.CChain()
```

## Setting up Wallet

A wallet is required for signing transactions with your private key and thus making it valid.

```
// For signing an unsigned transaction
const wallet = new ethers.Wallet(privateKey)
const address = wallet.address
```

## Function for Estimating Max Fee and Max Priority Fee

The function `calcFeeData()` estimates the max fee and max priority fee per gas according to network activity using the C-Chain APIs. This function returns max fee and max priority fee per gas in units of `nAVAX` or `gwei` ( $1 \text{ AVAX} = 10^{18} \text{ gwei}$ ).

```
// Function to estimate max fee and max priority fee
const calcFeeData = async (
 maxFeePerGas = undefined,
 maxPriorityFeePerGas = undefined
) => {
 const baseFee = parseInt(await cchain.getBaseFee(), 16) / 1e9
 maxPriorityFeePerGas =
 maxPriorityFeePerGas == undefined
 ? parseInt(await cchain.getMaxPriorityFeePerGas(), 16) / 1e9
 : maxPriorityFeePerGas
 maxFeePerGas =
 maxFeePerGas == undefined ? baseFee + maxPriorityFeePerGas : maxFeePerGas

 if (maxFeePerGas < maxPriorityFeePerGas) {
 throw "Error: Max fee per gas cannot be less than max priority fee per gas"
 }

 return {
 maxFeePerGas: maxFeePerGas.toString(),
 maxPriorityFeePerGas: maxPriorityFeePerGas.toString(),
 }
}
```

Actual API returns base fee and priority fee in units of `wei` which is one-billionth of a billionth of `AVAX` ( $1 \text{ AVAX} = 10^{18} \text{ wei}$ ).

## Function to Create, Sign and Send Transaction

The function `sendAvax()` takes 4 arguments -

- `amount` - Amount of AVAX to send in the transaction
- `address` - Destination address to which we want to send AVAX
- `maxFeePerGas` - Desired maximum fee per gas you want to pay in nAVAX
- `maxPriorityFeePerGas` - Desired maximum priority fee per gas you want to pay in nAVAX
- `nonce` - Used as a differentiator for more than 1 transaction with same signer

The last 3 arguments are optional, and if `undefined` is passed, then it will use the `calcFeeData()` function to estimate them. Each transaction with the same data and parameters is differentiated by a nonce value. If there are more than 1 transactions with the same nonce signed by the same address, then only 1 of them with the highest effective priority fee will be accepted. `nonce` parameter should only be used when you are either re-issuing or cancelling a stuck transaction.

```
// Function to send AVAX
const sendAvax = async (
 amount,
 to,
 maxFeePerGas = undefined,
 maxPriorityFeePerGas = undefined,
 nonce = undefined
```

```

) => {
 if (nonce == undefined) {
 nonce = await HTTPSPProvider.getTransactionCount(address)
 }

 // If the max fee or max priority fee is not provided, then it will automatically calculate using CChain APIs
 ;({ maxFeePerGas, maxPriorityFeePerGas } = await calcFeeData(
 maxFeePerGas,
 maxPriorityFeePerGas
))

 maxFeePerGas = ethers.utils.parseUnits(maxFeePerGas, "gwei")
 maxPriorityFeePerGas = ethers.utils.parseUnits(maxPriorityFeePerGas, "gwei")

 // Type 2 transaction is for EIP1559
 const tx = {
 type: 2,
 nonce,
 to,
 maxPriorityFeePerGas,
 maxFeePerGas,
 value: ethers.utils.parseEther(amount),
 chainId,
 }

 tx.gasLimit = await HTTPSPProvider.estimateGas(tx)

 const signedTx = await wallet.signTransaction(tx)
 const txHash = ethers.utils.keccak256(signedTx)

 console.log("Sending signed transaction")

 // Sending a signed transaction and waiting for its inclusion
 await (await HTTPSPProvider.sendTransaction(signedTx)).wait()

 console.log(
 `View transaction with nonce ${nonce}: https://testnet.snowtrace.io/tx/${txHash}`
)
}
}

```

This function calculates transaction hash from the signed transaction and logs on the console, the URL for transaction status on the Snowtrace explorer.

## Calling the `sendAVAX()` Function

There are various ways to call this function. We may or may not pass the optional arguments like max fee and max priority fee. It is recommended to set the max fee as the maximum price per gas that you are willing to pay for a transaction, no matter how high or low the base fee will be, as at max you will only be charged the provided max fee, along with a small priority fee above the base fee.

If you do not pass these arguments, then it will automatically estimate the max fee and priority fee from the network. For example, let's say, I want to pay 100 nAVAX per gas for a transaction and a small tip of 2 nAVAX, then we will call the following function.

```
// setting max fee as 100 and priority fee as 2
sendAvax("0.01", "0x856EA4B78947c3A5CD2256F85B2B147fEBDb7124", 100, 2)
```

**This function should not be used without a max fee per gas. As you will have to pay the estimated price, even if it is higher than your budget.**

There could be the following cases -

| Max Fee   | Max Priority Fee | Comment                                                                                                                                                                                                                                               |
|-----------|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| undefined | 2                | It will calculate the max fee by adding the provided priority fee with the estimated base fee. Take extra precaution here as the max fee will now be capped by <code>baseFee + priorityFee</code> , which can consume all the provided priority fees. |
| 100       | undefined        | It will estimate the priority fee and use the provided max fee. If the estimated priority fee is more than the provided max fee, then it throws an error.                                                                                             |
| undefined | undefined        | It will estimate the base fee and priority fee from the network, and will add both the values to calculate the max fee per gas. Again, you have to pay whatever will be estimated.                                                                    |

You will get the following output on the successful submission of the signed transactions. Using this URL you can view the status of your transaction on Snowtrace.

```
View transaction with nonce 25:
https://testnet.snowtrace.io/tx/0xd5b92b85beaf283fbabefb95c9a17a6b346a05b6f9687f2d6e421aa79243b35
```

## Reissuance of Stuck Transaction

Sometimes during high network activity, all transactions couldn't make it to the latest blocks for a long time, due to relatively lower effective tip than the other transactions in the pool. We can either re-issue the same transaction with a higher priority fee or cancel the transaction. To re-issue the stuck transaction, you can send a new one with same amount and data but higher priority fee and same nonce value as the stuck transaction. The transaction with lower effective tip will automatically be rejected (due to same nonce), and you do not need to worry about it. You can also cancel the stuck transaction, by keeping the amount to 0, with a higher priority fee and same nonce. Let's say, the above transaction with a nonce value of 25 has stuck. You can then re-issue a new transaction with same nonce, but higher priority fee this time.

```
// reissuing transaction with nonce 25
sendAvax("0.01", "0x856EA4B78947c3A5CD2256F85B2B147fEBDb7124", 100, 10, 25)

// cancelling transaction with nonce 25
sendAvax("0", "0x856EA4B78947c3A5CD2256F85B2B147fEBDb7124", 100, 10, 25)
```

## Conclusion

You have learned about creating, signing, and sending transactions with dynamic fee parameters to the C-Chain of Avalanche network using JavaScript. It also explained, how to re-issue or cancel a stuck transaction, by sending a transaction with the same nonce. This tutorial points out the recommended way for choosing max fee cap and max priority fee cap for transactions and can also work as a general guide for all the EVM-based chains.

## Tools and Utilities

There are a number of tools for managing your node and developing Subnets. This page lists the most popular ones, explains what they do and their intended usage.

### AvalancheGo Installer

AvalancheGo Installer is a shell (bash) script that installs AvalancheGo on a Linux computer. This script sets up full, running node in a matter of minutes with minimal user input required. This is convenient if you want to run the node as a service on a standalone Linux installation, for example to set up a (Subnet) validator, use the node as a private RPC server and similar uses. It also makes upgrading or reinstalling the nodes easy.

GitHub: <https://github.com/ava-labs/avalanche-docs/blob/master/scripts/avalanchego-installer.sh>

Document: [Run an Avalanche Node Using the Install Script](#)

If you want to run a node in a more complex environment, like in a docker or Kubernetes container, or as a part of an installation orchestrated using a tool like Terraform, this installer probably won't fit your purposes.

### Avalanche CLI

Avalanche CLI is a developer-centric command line tool that gives you access to everything Avalanche. Setting up a local network, creating a Subnet, customizing the Subnet/VM configuration - this is the tool to use. It is under rapid development, so check back for new versions with expanded functionality.

GitHub: <https://github.com/ava-labs/avalanche-cli>

Document: [Create an EVM Subnet on a Local Network](#)

### Avalanche Network Runner (ANR)

The Avalanche Network Runner (**ANR**) allows a user to define, create and interact with a network of Avalanche nodes. Networks created with **ANR** are temporary and get destroyed when the tool is stopped, so the purpose of the tool is to be used for local development and testing the code in the early stages, before you're ready to deploy on permanent infrastructure (testnet or Mainnet).

GitHub: <https://github.com/ava-labs/avalanche-network-runner>

Document: [Avalanche Network Runner](#)

### Avalanche Plugin Manager (APM)

Avalanche Plugin Manager (**APM**) is a command-line tool to manage virtual machines binaries on existing AvalancheGo instances. It enables to add/remove nodes to Subnets, upgrade the VM plugin binaries as new versions get released to the plugin repository.

GitHub: <https://github.com/ava-labs/apm>

#### avalanche-plugins-core

`avalanche-plugins-core` is plugin repository that ships with the `apm`. A plugin repository consists of a set of virtual machine and Subnet definitions that the `apm` consumes to allow users to quickly and easily download and manage VM binaries.

GitHub: <https://github.com/ava-labs/avalanche-plugins-core>

### Avalanche Ops

A single command to launch and configure network infrastructure (virtual machines or cloud instances) and installs Avalanche nodes from scratch allowing for various configuration requirements. Provisions all resources required to run a node or network with recommended setups (configurable). This tool is intended for quickly creating, testing and iterating over various Avalanche network infrastructure configurations for testing and simulation purposes. Use this to play with various setups and reproduce potential problems and issues with possible configurations.

GitHub: <https://github.com/ava-labs/avalanche-ops>

**description:** In order to prevent spam, transactions on Avalanche require the payment of a transaction fee. The fee is paid in AVAX. Find out more information here.

## Avalanche Transaction Fee

In order to prevent spam, transactions on Avalanche require the payment of a transaction fee. The fee is paid in AVAX. **The transaction fee is burned (destroyed forever).**

When you issue a transaction through Avalanche's API, the transaction fee is automatically deducted from one of the addresses you control.

### Fee Schedule

Different types of transactions require payment of a different transaction fee. This table shows the transaction fee schedule:

| Chain | Transaction Type    | Transaction Fee (AVAX) |
|-------|---------------------|------------------------|
| P     | : Create Subnet     | 1                      |
| P     | : Create Blockchain | 1                      |
| P     | : Add Validator     | 0                      |
| P     | : Add Delegator     | 0                      |
| P     | : Import AVAX       | 0.001                  |
| P     | : Export AVAX       | 0.001                  |
| X     | : Send              | 0.001                  |
| X     | : Create Asset      | 0.01                   |
| X     | : Mint Asset        | 0.001                  |
| X     | : Import AVAX       | 0.001                  |
| X     | : Export AVAX       | 0.001                  |
| C     | : Simple send       | >= 0.001575*           |

(\*) C-Chain gas price varies. See below.

### C-Chain Fees

The Avalanche C-Chain uses an algorithm to determine the "base fee" for a transaction. The base fee increases when network utilization is above the target utilization and decreases when network utilization is below the target.

#### Dynamic Fee Transactions

Transaction fees for non-atomic transactions are based on Ethereum's EIP-1559 style Dynamic Fee Transactions, which consists of a gas fee cap and a gas tip cap.

The fee cap specifies the maximum price the transaction is willing to pay per unit of gas. The tip cap (also called the priority fee) specifies the maximum amount above the base fee that the transaction is willing to pay per unit of gas. Therefore, the effective gas price paid by a transaction will be `min(gasFeeCap, baseFee + gasTipCap)`. Unlike in Ethereum, where the priority fee is paid to the miner that produces the block, in Avalanche both the base fee and the priority fee are burned. For legacy transactions, which only specify a single gas price, the gas price serves as both the gas fee cap and the gas tip cap.

Use the `eth_baseFee` API method to estimate the base fee for the next block. If more blocks are produced in between the time that you construct your transaction and it is included in a block, the base fee could be different from the base fee estimated by the API call, so it is important to treat this value as an estimate.

Next, use `eth_maxPriorityFeePerGas` API call to estimate the priority fee needed to be included in a block. This API call will look at the most recent blocks and see what tips have been paid by recent transactions in order to be included in the block.

Transactions are ordered by the priority fee, then the timestamp (oldest first).

Based off of this information, you can specify the `gasFeeCap` and `gasTipCap` to your liking based on how you prioritize getting your transaction included as quickly as possible vs. minimizing the price paid per unit of gas.

#### Base Fee

The base fee can go as low as 25 nAVAX (Gwei) and has no upper bound. You can use the `eth_baseFee` and `eth_maxPriorityFeePerGas` API methods, or [Snowtrace's C-Chain Gas Tracker](#), to estimate the gas price to use in your transactions.

## Further Readings

- [Adjusting Gas Price During High Network Activity](#)
- [Sending Transactions with Dynamic Fees using JavaScript](#)

## Atomic Transaction Fees

C-Chain atomic transactions (that is imports and exports from/to other chains) charge dynamic fees based on the amount of gas used by the transaction and the base fee of the block that includes the atomic transaction.

Gas Used:

```
+-----+-----+
| Item : Gas |
+-----+-----+
| Unsigned Tx Byte : 1 |
+-----+-----+
| Signature : 1000 |
+-----+-----+
| Per Atomic Tx : 10000 |
+-----+-----+
```

Therefore, the gas used by an atomic transaction is `1 * len(unsignedTxBytes) + 1,000 * numSignatures + 10,000`

The TX fee additionally takes the base fee into account. Due to the fact that atomic transactions use units denominated in 9 decimal places, the base fee must be converted to 9 decimal places before calculating the actual fee paid by the transaction. Therefore, the actual fee is: `gasUsed * baseFee` (converted to 9 decimals).

## Abigen

`abigen` is a tool provided by the Go Ethereum (Geth) client that generates Go bindings for Solidity smart contracts. Developers would need to use abigen when they want to interact with a smart contract written in Solidity from a Go programming language application. It enables developers to easily call functions and access data from Solidity contracts in a Go application. This tutorial demonstrates how to compile a solidity contract into Golang to deploy and call contracts programmatically.

## How to Build

Download the solidity compiler from [solc-bin](#).

Copy the appropriate compiler into your current path. `~/bin` is a common path in most Linux distributions.

```
cp linux-amd64/solc-linux-amd64-v0.8.9+commit.e5eed63a ~/bin
```

Ensure solc can run.

```
solc --version
solc, the solidity compiler commandline interface
Version: 0.8.9+commit.e5eed63a.Linux.g++
```

Build Abigen.

```
cd ~/go/src/github.com/ava-labs/avalanche
go build -o abigen cmd/abigen/main.go
cp abigen ~/bin
```

Compile a contract.

```
abigen --sol counter.sol --pkg main --out counter.go
```

This will produce `counter.go` suitable to interact with contract.

## Example Code

Setup the connection to `avalanchego`, then deploy, call, and fetch values from the contract.

Abigen offers more features for complicated contracts, the following is provided as an example to get started using the basic contract

```
package main

import (
 "context"
 "log"
 "math/big"
 "strings"
 "time"
```

```

"github.com/ava-labs/avalanche-go/utils/constants"
"github.com/ava-labs/avalanche-go/utils/formatting"
"github.com/ava-labs/coreth/accounts/abi/bind"
"github.com/ava-labs/coreth/core/types"
"github.com/ava-labs/coreth/ethclient"
"github.com/ava-labs/coreth/params"
"github.com/ava-labs/coreth/rpc"
"github.com/decred/dcrd/dcrec/secp256k1/v3"
"github.com/ethereum/go-ethereum/common"
"github.com/ethereum/go-ethereum/crypto"
)

func main() {
 // setup client
 rc, err := rpc.Dial("http://localhost:9650/ext/bc/C/rpc")
 if err != nil {
 log.Fatal(err)
 }
 ec := ethclient.NewClient(rc)

 ctx := context.Background()

 // fetch networkid
 networkId, err := ec.ChainID(ctx)
 if err != nil {
 log.Fatal(err)
 }

 // parse key
 privateKeyString := "PrivateKey-ewoqjP7PxI4yr3iLTpLisriqt94hdyDFNgchSxGGztUrTXtNN"
 privateKeyBytes, err := formatting.Decode(formatting.CB58, strings.TrimPrefix(privateKeyString, constants.SecretKeyPrefix))
 if err != nil {
 log.Fatal(err)
 }
 privateKey := secp256k1.PrivateKeyFromBytes(privateKeyBytes)
 privateKeyECDSA := privateKey.ToECDSA()

 // derive 'c' address
 cAddress := crypto.PubkeyToAddress(privateKeyECDSA.PublicKey)

 // setup signer and transaction options.
 signer := types.LatestSignerForChainID(networkId)
 to := &bind.TransactOpts{
 Signer: func(address common.Address, transaction *types.Transaction) (*types.Transaction, error) {
 return types.SignTx(transaction, signer, privateKeyECDSA)
 },
 From: cAddress,
 Context: ctx,
 GasLimit: params.ApricotPhase1GasLimit,
 }

 // deploy the contract
 storageAddress, storageTransaction, storageContract, err := DeployStorage(to, ec)
 if err != nil {
 log.Fatal(err)
 }

 // wait for the transaction to be accepted
 for {
 r, err := ec.TransactionReceipt(ctx, storageTransaction.Hash())
 if err != nil {
 if err.Error() != "not found" {
 log.Fatal(err)
 }
 time.Sleep(1 * time.Second)
 continue
 }
 if r.Status != 0 {
 break
 }
 time.Sleep(1 * time.Second)
 }

 log.Println("storageAddress", storageAddress)
 log.Println("storageTransaction", storageTransaction)

 // Call store on the contract
 storeTransaction, err := storageContract.Store(to, big.NewInt(1), common.BytesToAddress([]byte("addr1")))
}

```

```

if err != nil {
 log.Fatal(err)
}

// wait for the transaction
for {
 r, err := ec.TransactionReceipt(ctx, storeTransaction.Hash())
 if err != nil {
 if err.Error() != "not found" {
 log.Fatal(err)
 }
 time.Sleep(1 * time.Second)
 continue
 }
 if r.Status != 0 {
 break
 }
 time.Sleep(1 * time.Second)
}

log.Println("storeTransaction", storeTransaction)

// setup call options for storage
co := &bind.CallOpts{
 Accepted: true,
 Context: ctx,
 From: storageAddress,
}

// retrieve the value of the contract
storageValue, err := storageContract.Retrieve(co)
if err != nil {
 log.Fatal(err)
}

log.Println("storageValue", storageValue)
}

```

## AVM Transaction Format

This file is meant to be the single source of truth for how we serialize transactions in the Avalanche Virtual Machine (AVM). This document uses the [primitive serialization](#) format for packing and [secp256k1](#) for cryptographic user identification.

### Codec ID

Some data is prepended with a codec ID (uint16) that denotes how the data should be deserialized. Right now, the only valid codec ID is 0 ( 0x00 0x00 ).

### Transferable Output

Transferable outputs wrap an output with an asset ID.

#### What Transferable Output Contains

A transferable output contains an `AssetID` and an [Output](#).

- `AssetID` is a 32-byte array that defines which asset this output references.
- `Output` is an output, as defined [below](#). Outputs have four possible types: [SECP256K1TransferOutput](#), [SECP256K1MintOutput](#), [NFTTransferOutput](#) and [NFTMintOutput](#).

#### Gantt Transferable Output Specification

|                         |                    |
|-------------------------|--------------------|
| +-----+-----+-----+     |                    |
| asset_id : [32]byte     | 32 bytes           |
| +-----+-----+-----+     |                    |
| output : Output         | size(output) bytes |
| +-----+-----+-----+     |                    |
| 32 + size(output) bytes |                    |
| +-----+-----+-----+     |                    |

#### Proto Transferable Output Specification

```

message TransferableOutput {
 bytes asset_id = 1; // 32 bytes
}

```

```

 Output output = 2; // size(output)
}

```

### Transferable Output Example

Let's make a transferable output:

- AssetID : 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- Output : "Example SECP256K1 Transfer Output from below"

```

[
 AssetID <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
 Output <-
0x00000007000000000000303900000000000d4310000001000000251025c61fbfcfc078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c89<
]
=
[
 // assetID:
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 // output:
 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
 0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
 0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]

```

### Transferable Input

Transferable inputs describe a specific UTXO with a provided transfer input.

#### What Transferable Input Contains

A transferable input contains a `TxID`, `UTXOIndex`, `AssetID` and an `Input`.

- `TxID` is a 32-byte array that defines which transaction this input is consuming an output from. Transaction IDs are calculated by taking sha256 of the bytes of the signed transaction.
- `UTXOIndex` is an int that defines which UTXO this input is consuming in the specified transaction.
- `AssetID` is a 32-byte array that defines which asset this input references.
- `Input` is an input, as defined below. This can currently only be a [SECP256K1 transfer input](#)

#### Gantt Transferable Input Specification

|            |            |                        |
|------------|------------|------------------------|
| tx_id      | : [32]byte | 32 bytes               |
| utxo_index | : int      | 04 bytes               |
| asset_id   | : [32]byte | 32 bytes               |
| input      | : Input    | size(input) bytes      |
|            |            | 68 + size(input) bytes |

#### Proto Transferable Input Specification

```

message TransferableInput {
 bytes tx_id = 1; // 32 bytes
 uint32 utxo_index = 2; // 04 bytes
 bytes asset_id = 3; // 32 bytes
 Input input = 4; // size(input)
}

```

### Transferable Input Example

Let's make a transferable input:

- TxID : 0xf1e1d1c1bia191817161514131211101f0e0d0c0b0a090807060504030201000

- `UTXOIndex` : 5
- `AssetID` : 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- `Input` : "Example SECP256K1 Transfer Input from below"

```
[
 TxID <- 0xf1e1d1c1b1a191817161514131211101f0e0d0c0b0a090807060504030201000
 UTXOIndex <- 0x0000005
 AssetID <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
 Input <- 0x00000005000000000075bcd15000000020000000700000003
]
=
[
 // txID:
 0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
 0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
 0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
 0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
 // utxoIndex:
 0x00, 0x00, 0x00, 0x05,
 // assetID:
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 // input:
 0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
 0x07, 0x5b, 0xcd, 0x15, 0x00, 0x00, 0x00, 0x02,
 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x07
]
```

## Transferable Op

Transferable operations describe a set of UTXOs with a provided transfer operation. Only one Asset ID is able to be referenced per operation.

### What Transferable Op Contains

A transferable operation contains an `AssetID`, `UTXOIDs`, and a `TransferOp`.

- `AssetID` is a 32-byte array that defines which asset this operation changes.
- `UTXOIDs` is an array of TxID-OutputIndex tuples. This array must be sorted in lexicographical order.
- `TransferOp` is a [transferable operation object](#).

### Gantt Transferable Op Specification

|                           |                              |
|---------------------------|------------------------------|
| asset_id : [32]byte       | 32 bytes                     |
| +-----+-----+             | +-----+                      |
| utxo_ids : []UTXOID       | 4 + 36 * len(utxo_ids) bytes |
| +-----+-----+             | +-----+                      |
| transfer_op : TransferOp  | size(transfer_op) bytes      |
| +-----+-----+             | +-----+                      |
| 36 + 36 * len(utxo_ids)   |                              |
| + size(transfer_op) bytes |                              |
| +-----+-----+             | +-----+                      |

### Proto Transferable Op Specification

```
message UTXOID {
 bytes tx_id = 1; // 32 bytes
 uint32 utxo_index = 2; // 04 bytes
}
message TransferableOp {
 bytes asset_id = 1; // 32 bytes
 repeated UTXOID utxo_ids = 2; // 4 + 36 * len(utxo_ids) bytes
 TransferOp transfer_op = 3; // size(transfer_op)
}
```

## Transferable Op Example

Let's make a transferable operation:

- `AssetID` : 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- `UTXOIDs` :
  - `UTXOID` :
    - `TxID` : 0xf1e1d1c1b1a191817161514131211101f0e0d0c0b0a090807060504030201000

```

■ UTXOIndex : 5

• Op : "Example Transfer Op from below"

[

AssetID <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
UTXOIDs <- [
{
 TxID:0xf1d1c1b1a191817161514131211101f0e0d0c0b0a090807060504030201000
 UTXOIndex:5
}
]
Op <-
0x0000000d000000200000030000007000303900000034311000000001000000251025c61fbfcfc078f69334e834be6dd26d55a955c3344128e060128ede:
]
=
[
// assetID:
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
// number of utxoIDs:
0x00, 0x00, 0x00, 0x01,
// txID:
0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
// utxoIndex:
0x00, 0x00, 0x00, 0x05,
// op:
0x00, 0x00, 0x00, 0x0d, 0x00, 0x00, 0x00, 0x02,
0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x07,
0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x03,
0x43, 0x11, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00,
0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61, 0xfb,
0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0x8f, 0x34,
0xbe, 0x6d, 0xd2, 0xed, 0x55, 0xa9, 0x55, 0xc3,
0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e, 0xde,
0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89, 0x43,
0xab, 0x08, 0x59,
]
]

```

## Outputs

Outputs have four possible types: [SECP256K1TransferOutput](#), [SECP256K1MintOutput](#), [NFTTransferOutput](#) and [NFTMintOutput](#).

### SECP256K1 Mint Output

A [secp256k1](#) mint output is an output that is owned by a collection of addresses.

#### What SECP256K1 Mint Output Contains

A secp256k1 Mint output contains a `TypeID`, `Locktime`, `Threshold`, and `Addresses`.

- `TypeID` is the ID for this output type. It is `0x00000006`.
- `Locktime` is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- `Threshold` is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`. If `Addresses` is empty, must be 0.
- `Addresses` is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

#### Gantt SECP256K1 Mint Output Specification

|                                         |                                                 |
|-----------------------------------------|-------------------------------------------------|
| <code>  type_id : int  </code>          | <code>4 bytes  </code>                          |
| <code>+-----+-----+</code>              | <code>-----+----- </code>                       |
| <code>  locktime : long  </code>        | <code>8 bytes  </code>                          |
| <code>+-----+-----+</code>              | <code>-----+----- </code>                       |
| <code>  threshold : int  </code>        | <code>4 bytes  </code>                          |
| <code>+-----+-----+</code>              | <code>-----+----- </code>                       |
| <code>  addresses : [][20]byte  </code> | <code>4 + 20 * len(addresses) bytes  </code>    |
| <code>+-----+-----+</code>              | <code>-----+----- </code>                       |
|                                         | <code>  20 + 20 * len(addresses) bytes  </code> |
|                                         | <code>-----+----- </code>                       |

## Proto SECP256K1 Mint Output Specification

```
message SECP256K1MintOutput {
 uint32 typeID = 1; // 04 bytes
 uint64 locktime = 2; // 08 bytes
 uint32 threshold = 3; // 04 bytes
 repeated bytes addresses = 4; // 04 bytes + 20 bytes * len(addresses)
}
```

## SECP256K1 Mint Output Example

Let's make a SECP256K1 mint output with:

- **TypeID** : 6
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
- 0x51025c61fbfcfc078f69334f834be6dd26d55a955
- 0xc3344128e060128ede3523a24a461c8943ab0859

```
[
 TypeID <- 0x00000006
 Locktime <- 0x000000000000d431
 Threshold <- 0x00000001
 Addresses <- [
 0x51025c61fbfcfc078f69334f834be6dd26d55a955,
 0xc3344128e060128ede3523a24a461c8943ab0859,
]
]
=
[
 // typeID:
 0x00, 0x00, 0x00, 0x06,
 // locktime:
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
 // threshold:
 0x00, 0x00, 0x00, 0x01,
 // number of addresses:
 0x00, 0x00, 0x00, 0x02,
 // addrs[0]:
 0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
 0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,
 0x6d, 0x55, 0xa9, 0x55,
 // addrs[1]:
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]
```

## SECP256K1 Transfer Output

A [secp256k1](#) transfer output allows for sending a quantity of an asset to a collection of addresses after a specified Unix time.

### What SECP256K1 Transfer Output Contains

A secp256k1 transfer output contains a `TypeID`, `Amount`, `Locktime`, `Threshold`, and `Addresses`.

- **TypeID** is the ID for this output type. It is `0x00000007`.
- **Amount** is a long that specifies the quantity of the asset that this output owns. Must be positive.
- **Locktime** is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- **Threshold** is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`. If `Addresses` is empty, must be 0.
- **Addresses** is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

### Gantt SECP256K1 Transfer Output Specification

|                    |         |  |
|--------------------|---------|--|
| -----+-----+-----+ |         |  |
| type_id : int      | 4 bytes |  |
| -----+-----+-----+ |         |  |
| amount : long      | 8 bytes |  |
| -----+-----+-----+ |         |  |
| locktime : long    | 8 bytes |  |
| -----+-----+-----+ |         |  |
| threshold : int    | 4 bytes |  |

```
+-----+-----+
| addresses : [] [20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+
| 28 + 20 * len(addresses) bytes |
+-----+
```

### Proto SECP256K1 Transfer Output Specification

```
message SECP256K1TransferOutput {
 uint32 typeID = 1; // 04 bytes
 uint64 amount = 2; // 08 bytes
 uint64 locktime = 3; // 08 bytes
 uint32 threshold = 4; // 04 bytes
 repeated bytes addresses = 5; // 04 bytes + 20 bytes * len(addresses)
}
```

### SECP256K1 Transfer Output Example

Let's make a secp256k1 transfer output with:

- **TypeID** : 7
- **Amount** : 12345
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
- 0x51025c61fbfcfc078f69334f834be6dd26d55a955
- 0xc3344128e060128ede3523a24a461c8943ab0859

```
[
 TypeID <- 0x00000007
 Amount <- 0x00000000000003039
 Locktime <- 0x000000000000d431
 Threshold <- 0x00000001
 Addresses <- [
 0x51025c61fbfcfc078f69334f834be6dd26d55a955,
 0xc3344128e060128ede3523a24a461c8943ab0859,
]
]
=
[
 // typeID:
 0x00, 0x00, 0x00, 0x07,
 // amount:
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x39,
 // locktime:
 0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
 // threshold:
 0x00, 0x00, 0x00, 0x01,
 // number of addresses:
 0x00, 0x00, 0x00, 0x02,
 // addrs[0]:
 0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
 0xf6, 0x93, 0x34, 0x8f, 0x34, 0xbe, 0x6d, 0xd2,
 0x6d, 0x55, 0xa9, 0x55,
 // addrs[1]:
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]
```

### NFT Mint Output

An NFT mint output is an NFT that is owned by a collection of addresses.

#### What NFT Mint Output Contains

An NFT Mint output contains a `TypeID` , `GroupID` , `Locktime` , `Threshold` , and `Addresses` .

- **TypeID** is the ID for this output type. It is `0x0000000a` .
- **GroupID** is an int that specifies the group this NFT is issued to.
- **Locktime** is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- **Threshold** is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses` . If `Addresses` is empty, must be 0.
- **Addresses** is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

## Gantt NFT Mint Output Specification

```
+-----+-----+-----+
| type_id : int | 4 bytes |
+-----+-----+-----+
| group_id : int | 4 bytes |
+-----+-----+-----+
| locktime : long | 8 bytes |
+-----+-----+-----+
| threshold : int | 4 bytes |
+-----+-----+-----+
| addresses : [][20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+-----+
| 24 + 20 * len(addresses) bytes |
+-----+-----+
```

## Proto NFT Mint Output Specification

```
message NFTMintOutput {
 uint32 typeID = 1; // 04 bytes
 uint32 group_id = 2; // 04 bytes
 uint64 locktime = 3; // 08 bytes
 uint32 threshold = 4; // 04 bytes
 repeated bytes addresses = 5; // 04 bytes + 20 bytes * len(addresses)
}
```

## NFT Mint Output Example

Let's make an NFT mint output with:

- **TypeID** : 10
- **GroupID** : 12345
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
  - 0x51025c61fbfcfc078f69334f834be6dd26d55a955
  - 0xc3344128e060128ede3523a24a461c8943ab0859

```
[
 TypeID <- 0x0000000a
 GroupID <- 0x00003039
 Locktime <- 0x000000000000d431
 Threshold <- 0x00000001
 Addresses <- [
 0x51025c61fbfcfc078f69334f834be6dd26d55a955,
 0xc3344128e060128ede3523a24a461c8943ab0859,
]
]
=
[
 // TypeID
 0x00, 0x00, 0x00, 0x0a,
 // groupID:
 0x00, 0x00, 0x30, 0x39,
 // locktime:
 0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
 // threshold:
 0x00, 0x00, 0x00, 0x01,
 // number of addresses:
 0x00, 0x00, 0x00, 0x02,
 // addrs[0]:
 0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
 0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,
 0x6d, 0x55, 0xa9, 0x55,
 // addrs[1]:
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]
```

## NFT Transfer Output

An NFT transfer output is an NFT that is owned by a collection of addresses.

## What NFT Transfer Output Contains

An NFT transfer output contains a `TypeID`, `GroupID`, `Payload`, `Locktime`, `Threshold`, and `Addresses`.

- `TypeID` is the ID for this output type. It is `0x0000000b`.
- `GroupID` is an int that specifies the group this NFT was issued with.
- `Payload` is an arbitrary string of bytes no longer than 1024 bytes.
- `Locktime` is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- `Threshold` is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`. If `Addresses` is empty, must be 0.
- `Addresses` is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

## Gantt NFT Transfer Output Specification

|                                                         |                                              |
|---------------------------------------------------------|----------------------------------------------|
| <code>  type_id : int  </code>                          | <code>4 bytes  </code>                       |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>  group_id : int  </code>                         | <code>4 bytes  </code>                       |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>  payload : []byte  </code>                       | <code>4 + len(payload) bytes  </code>        |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>  locktime : long  </code>                        | <code>8 bytes  </code>                       |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>  threshold : int  </code>                        | <code>4 bytes  </code>                       |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>  addresses : [[20]byte  </code>                  | <code>4 + 20 * len(addresses) bytes  </code> |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |
| <code>             28 + len(payload)  </code>           | <code> </code>                               |
| <code>             + 20 * len(addresses) bytes  </code> | <code>+</code>                               |
| <code>+-----+-----+</code>                              | <code>+-----+</code>                         |

## Proto NFT Transfer Output Specification

```
message NFTTransferOutput {
 uint32 typeID = 1; // 04 bytes
 uint32 group_id = 2; // 04 bytes
 bytes payload = 3; // 04 bytes + len(payload)
 uint64 locktime = 4; // 08 bytes
 uint32 threshold = 5; // 04 bytes
 repeated bytes addresses = 6; // 04 bytes + 20 bytes * len(addresses)
}
```

## NFT Transfer Output Example

Let's make an NFT transfer output with:

- `TypeID` : 11
- `GroupID` : 12345
- `Payload` : NFT Payload
- `Locktime` : 54321
- `Threshold` : 1
- `Addresses` :
- `0x51025c61fbfcfc078f69334f834be6dd26d55a955`
- `0xc3344128e060128ede3523a24a461c8943ab0859`

```
[
 TypeID <- 0x0000000b
 GroupID <- 0x00003039
 Payload <- 0x4e4654205061796c6f6164
 Locktime <- 0x0000000000000d431
 Threshold <- 0x00000001
 Addresses <- [
 0x51025c61fbfcfc078f69334f834be6dd26d55a955,
 0xc3344128e060128ede3523a24a461c8943ab0859,
]
]
=
[
 // TypeID:
 0x00, 0x00, 0x00, 0x0b,
 // groupID:
 0x00, 0x00, 0x30, 0x39,
 // length of payload:
 0x00, 0x00, 0x00, 0x0b,
```

```

// payload:
0x4e, 0x46, 0x54, 0x20, 0x50, 0x61, 0x79, 0x6c,
0x6f, 0x61, 0x64,
// locktime:
0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
// threshold:
0x00, 0x00, 0x00, 0x01,
// number of addresses:
0x00, 0x00, 0x00, 0x02,
// addrs[0]:
0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,
0xed, 0x55, 0xa9, 0x55,
// addrs[1]:
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
]

```

## Inputs

Inputs have one possible type: `SECP256K1TransferInput`.

### SECP256K1 Transfer Input

A [secp256k1](#) transfer input allows for spending an unspent secp256k1 transfer output.

#### What SECP256K1 Transfer Input Contains

A secp256k1 transfer input contains an `Amount` and `AddressIndices`.

- `TypeID` is the ID for this input type. It is `0x00000005`.
- `Amount` is a long that specifies the quantity that this input should be consuming from the UTXO. Must be positive. Must be equal to the amount specified in the UTXO.
- `AddressIndices` is a list of unique ints that define the private keys that are being used to spend the UTXO. Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.

#### Gantt SECP256K1 Transfer Input Specification

|                              |                      |                                                  |
|------------------------------|----------------------|--------------------------------------------------|
| <code>type_id</code>         | <code>: int</code>   | 4 bytes                                          |
| <code>amount</code>          | <code>: long</code>  | 8 bytes                                          |
| <code>address_indices</code> | <code>: []int</code> | <code>4 + 4 * len(address_indices)</code> bytes  |
|                              |                      | <code>16 + 4 * len(address_indices)</code> bytes |

#### Proto SECP256K1 Transfer Input Specification

```

message SECP256K1TransferInput {
 uint32 typeID = 1; // 04 bytes
 uint64 amount = 2; // 08 bytes
 repeated uint32 address_indices = 3; // 04 bytes + 04 bytes * len(address_indices)
}

```

#### SECP256K1 Transfer Input Example

Let's make a payment input with:

- `TypeId` : 5
- `Amount` : 123456789
- `AddressIndices` : [ 3 , 7 ]

```

[
 TypeID <- 0x00000005
 Amount <- 123456789 = 0x000000000075bcd15,
 AddressIndices <- [0x00000003, 0x00000007]
]
=
[
 // type id:
 0x00, 0x00, 0x00, 0x05,
 // amount:
]

```

```

0x00, 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,
// length:
0x00, 0x00, 0x00, 0x02,
// sig[0]
0x00, 0x00, 0x00, 0x03,
// sig[1]
0x00, 0x00, 0x00, 0x07,
]

```

## Operations

Operations have three possible types: `SECP256K1MintOperation`, `NFTMintOp`, and `NFTTransferOp`.

### SECP256K1 Mint Operation

A `secp256k1` mint operation consumes a SECP256K1 mint output, creates a new mint output and sends a transfer output to a new set of owners.

#### What SECP256K1 Mint Operation Contains

A `secp256k1` Mint operation contains a `TypeID`, `AddressIndices`, `MintOutput`, and `TransferOutput`.

- `TypeID` is the ID for this output type. It is `0x00000008`.
- `AddressIndices` is a list of unique ints that define the private keys that are being used to spend the [UTXO](#). Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.
- `MintOutput` is a [SECP256K1 Mint output](#).
- `TransferOutput` is a [SECP256K1 Transfer output](#).

#### Gantt SECP256K1 Mint Operation Specification

|                                                     |                                                   |
|-----------------------------------------------------|---------------------------------------------------|
| <code>  type_id : int  </code>                      | <code>4 bytes  </code>                            |
| <code>+-----+-----+</code>                          | <code>+-----+-----+</code>                        |
| <code>  address_indices : []int  </code>            | <code>4 + 4 * len(address_indices) bytes  </code> |
| <code>+-----+-----+</code>                          | <code>+-----+-----+</code>                        |
| <code>  mint_output : MintOutput  </code>           | <code>size(mint_output) bytes  </code>            |
| <code>+-----+-----+</code>                          | <code>+-----+-----+</code>                        |
| <code>  transfer_output : TransferOutput  </code>   | <code>size(transfer_output) bytes  </code>        |
| <code>+-----+-----+</code>                          | <code>+-----+-----+</code>                        |
| <code>       8 + 4 * len(address_indices)  </code>  | <code>      </code>                               |
| <code>       + size(mint_output)  </code>           | <code>      </code>                               |
| <code>       + size(transfer_output) bytes  </code> | <code>      </code>                               |
| <code>+-----+-----+</code>                          | <code>+-----+-----+</code>                        |

#### Proto SECP256K1 Mint Operation Specification

```

message SECP256K1MintOperation {
 uint32 typeID = 1; // 4 bytes
 repeated uint32 address_indices = 2; // 04 bytes + 04 bytes * len(address_indices)
 MintOutput mint_output = 3; // size(mint_output)
 TransferOutput transfer_output = 4; // size(transfer_output)
}

```

### SECP256K1 Mint Operation Example

Let's make a `secp256k1` mint operation with:

- `TypeId` : 8
- `AddressIndices` :
- 0x00000003
- 0x00000007
- `MintOutput` : "Example SECP256K1 Mint Output from above"
- `TransferOutput` : "Example SECP256K1 Transfer Output from above"

```

[
 TypeID <- 0x00000008
 AddressIndices <- [0x00000003, 0x00000007]
 MintOutput <-
0x0000000600000000000d4310000001000000251025c61fbfcf078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c89
 TransferOutput <-
0x00000007000000000000303900000000000d4310000001000000251025c61fbfcf078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c89
]
=
[
 // typeID
 0x00, 0x00, 0x00, 0x08,
]

```

```

// number of address_indices:
0x00, 0x00, 0x00, 0x02,
// address_indices[0]:
0x00, 0x00, 0x00, 0x03,
// address_indices[1]:
0x00, 0x00, 0x00, 0x07,
// mint output
0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
// transfer output
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
]

```

## NFT Mint Op

An NFT mint operation consumes an NFT mint output and sends an unspent output to a new set of owners.

### What NFT Mint Op Contains

An NFT mint operation contains a `TypeID`, `AddressIndices`, `GroupID`, `Payload`, and `Output` of addresses.

- `TypeID` is the ID for this operation type. It is `0x0000000c`.
- `AddressIndices` is a list of unique ints that define the private keys that are being used to spend the UTXO. Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.
- `GroupID` is an int that specifies the group this NFT is issued to.
- `Payload` is an arbitrary string of bytes no longer than 1024 bytes.
- `Output` is not a `TransferableOutput`, but rather is a lock time, threshold, and an array of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

### Gantt NFT Mint Op Specification

|                              |                         |                                                     |
|------------------------------|-------------------------|-----------------------------------------------------|
| <code>type_id</code>         | <code>: int</code>      | 4 bytes                                             |
| <code>address_indices</code> | <code>: []int</code>    | $4 + 4 * \text{len}(\text{address\_indices})$ bytes |
| <code>group_id</code>        | <code>: int</code>      | 4 bytes                                             |
| <code>payload</code>         | <code>: []byte</code>   | $4 + \text{len}(\text{payload})$ bytes              |
| <code>outputs</code>         | <code>: []Output</code> | $4 + \text{size}(\text{outputs})$ bytes             |
|                              |                         | $20 +$                                              |
|                              |                         | $4 * \text{len}(\text{address\_indices}) +$         |
|                              |                         | $\text{len}(\text{payload}) +$                      |
|                              |                         | $\text{size}(\text{outputs})$ bytes                 |

### Proto NFT Mint Op Specification

```

message NFTMintOp {
 uint32 typeID = 1; // 04 bytes
 repeated uint32 address_indices = 2; // 04 bytes + 04 bytes * len(address_indices)
 uint32 group_id = 3; // 04 bytes
 bytes payload = 4; // 04 bytes + len(payload)
 repeated bytes outputs = 5; // 04 bytes + size(outputs)
}

```

### NFT Mint Op Example

Let's make an NFT mint operation with:

- **TypeID** : 12
- **AddressIndices** :
  - 0x00000003
  - 0x00000007
- **GroupID** : 12345
- **Payload** : 0x431100
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
- 0xc3344128e060128ede3523a24a461c8943ab0859

```
[
 TypeID <- 0x0000000c
 AddressIndices <- [
 0x00000003,
 0x00000007,
]
 GroupID <- 0x00003039
 Payload <- 0x431100
 Locktime <- 0x000000000000d431
 Threshold <- 0x00000001
 Addresses <- [
 0xc3344128e060128ede3523a24a461c8943ab0859
]
]
=
[
 // Type ID
 0x00, 0x00, 0x00, 0x0c,
 // number of address indices:
 0x00, 0x00, 0x00, 0x02,
 // address index 0:
 0x00, 0x00, 0x00, 0x03,
 // address index 1:
 0x00, 0x00, 0x00, 0x07,
 // groupID:
 0x00, 0x00, 0x30, 0x39,
 // length of payload:
 0x00, 0x00, 0x00, 0x03,
 // payload:
 0x43, 0x11, 0x00,
 // number of outputs:
 0x00, 0x00, 0x00, 0x01,
 // outputs[0]
 // locktime:
 0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
 // threshold:
 0x00, 0x00, 0x00, 0x01,
 // number of addresses:
 0x00, 0x00, 0x00, 0x01,
 // addrs[0]:
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]
]
```

## NFT Transfer Op

An NFT transfer operation sends an unspent NFT transfer output to a new set of owners.

### What NFT Transfer Op Contains

An NFT transfer operation contains a `TypeID`, `AddressIndices` and an untyped `NFTTransferOutput`.

- `TypeID` is the ID for this output type. It is `0x0000000d`.
- `AddressIndices` is a list of unique ints that define the private keys that are being used to spend the UTXO. Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.
- `NFTTransferOutput` is the output of this operation and must be an [NFT Transfer Output](#). This output doesn't have the `TypeID`, because the type is known by the context of being in this operation.

### Gantt NFT Transfer Op Specification

|               |         |
|---------------|---------|
| +-----+-----+ |         |
| type_id : int | 4 bytes |
| +-----+-----+ |         |

```

| address_indices : []int | 4 + 4 * len(address_indices) bytes |
+-----+-----+
| group_id : int | 4 bytes |
+-----+-----+
| payload : []byte | 4 + len(payload) bytes |
+-----+-----+
| locktime : long | 8 bytes |
+-----+-----+
| threshold : int | 4 bytes |
+-----+-----+
| addresses : [][]20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+
| 36 + len(payload) |
| + 4 * len(address_indices) |
| + 20 * len(addresses) bytes |
+-----+-----+

```

### Proto NFT Transfer Op Specification

```

message NFTTransferOp {
 uint32 typeID = 1; // 04 bytes
 repeated uint32 address_indices = 2; // 04 bytes + 04 bytes * len(address_indices)
 uint32 group_id = 3; // 04 bytes
 bytes payload = 4; // 04 bytes + len(payload)
 uint64 locktime = 5; // 08 bytes
 uint32 threshold = 6; // 04 bytes
 repeated bytes addresses = 7; // 04 bytes + 20 bytes * len(addresses)
}

```

### NFT Transfer Op Example

Let's make an NFT transfer operation with:

- **TypeID** : 13
- **AddressIndices** :
- 0x00000007
- 0x00000003
- **GroupID** : 12345
- **Payload** : 0x431100
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
- 0xc3344128e060128ede3523a24a461c8943ab0859
- 0x51025c61fbfcfc078f69334f834be6dd26d55a955

```

[
 TypeID <- 0x0000000d
 AddressIndices <- [
 0x00000007,
 0x00000003,
]
 GroupID <- 0x00003039
 Payload <- 0x431100
 Locktime <- 0x000000000000d431
 Threshold <- 00000001
 Addresses <- [
 0x51025c61fbfcfc078f69334f834be6dd26d55a955,
 0xc3344128e060128ede3523a24a461c8943ab0859,
]
]
=
[
 // Type ID
 0x00, 0x00, 0x00, 0x0d,
 // number of address indices:
 0x00, 0x00, 0x00, 0x02,
 // address index 0:
 0x00, 0x00, 0x00, 0x07,
 // address index 1:
 0x00, 0x00, 0x00, 0x03,
 // groupID:
 0x00, 0x00, 0x30, 0x39,
 // length of payload:
 0x00, 0x00, 0x00, 0x03,
 // payload:

```

```

0x43, 0x11, 0x00,
// locktime:
0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
// threshold:
0x00, 0x00, 0x00, 0x01,
// number of addresses:
0x00, 0x00, 0x00, 0x02,
// addrs[0]:
0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,
0x6d, 0x55, 0xa9, 0x55,
// addrs[1]:
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
]

```

## Initial State

Initial state describes the initial state of an asset when it is created. It contains the ID of the feature extension that the asset uses, and a variable length array of outputs that denote the genesis UTXO set of the asset.

### What Initial State Contains

Initial state contains a `FxID` and an array of `Output`.

- `FxID` is an int that defines which feature extension this state is part of. For SECP256K1 assets, this is `0x00000000`. For NFT assets, this is `0x00000001`.
- `Outputs` is a variable length array of `outputs`, as defined above.

### Gantt Initial State Specification

|                     |                         |
|---------------------|-------------------------|
| +-----+-----+-----+ |                         |
| fx_id : int         | 4 bytes                 |
| +-----+-----+-----+ |                         |
| outputs : []Output  | 4 + size(outputs) bytes |
| +-----+-----+-----+ |                         |
|                     | 8 + size(outputs) bytes |
| +-----+-----+-----+ |                         |

### Proto Initial State Specification

```

message InitialState {
 uint32 fx_id = 1; // 04 bytes
 repeated Output outputs = 2; // 04 + size(outputs) bytes
}

```

### Initial State Example

Let's make an initial state:

- `FxID` : `0x00000000`
- `InitialState` : `["Example SECP256K1 Transfer Output from above"]`

```

[
 FxID <- 0x00000000
 InitialState <- [
 0x00000007000000000000003039000000000000d43100000010000000251025c61fbfcf078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c894
]
]
=
[
 // fxID:
 0x00, 0x00, 0x00, 0x00,
 // num outputs:
 0x00, 0x00, 0x00, 0x01,
 // output:
 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
 0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0x8f,
 0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
]

```

```
 0x43, 0xab, 0x08, 0x59,
]
```

## Credentials

Credentials have two possible types: `SECP256K1Credential`, and `NFTCredential`. Each credential is paired with an Input or Operation. The order of the credentials match the order of the inputs or operations.

### SECP256K1 Credential

A `secp256k1` credential contains a list of 65-byte recoverable signatures.

#### What SECP256K1 Credential Contains

- `TypeID` is the ID for this type. It is `0x00000009`.
- `Signatures` is an array of 65-byte recoverable signatures. The order of the signatures must match the input's signature indices.

#### Gantt SECP256K1 Credential Specification

```
+-----+-----+
| type_id : int | 4 bytes |
+-----+-----+
| signatures : [] [65]byte | 4 + 65 * len(signatures) bytes |
+-----+-----+
| 8 + 65 * len(signatures) bytes |
+-----+
```

#### Proto SECP256K1 Credential Specification

```
message SECP256K1Credential {
 uint32 typeID = 1; // 4 bytes
 repeated bytes signatures = 2; // 4 bytes + 65 bytes * len(signatures)
}
```

### SECP256K1 Credential Example

Let's make a payment input with:

```
• TypeID : 9
• signatures :
• 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f303132333435363738393a3b3c3d3e
• 0x404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5e5d5f606162636465666768696a6b6c6e6d6f707172737475767778797a7b7c7d7e

[
 TypeID <- 0x00000009
 Signatures <- [

0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f303132333435363738393a3b3c3d3e3f(

0x404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5e5d5f606162636465666768696a6b6c6e6d6f707172737475767778797a7b7c7d7e7f(
]
]
=
[
 // Type ID
 0x00, 0x00, 0x00, 0x09,
 // length:
 0x00, 0x00, 0x00, 0x02,
 // sig[0]
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1e, 0x1d, 0x1f,
 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
 0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2e, 0x2d, 0x2f,
 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
 0x00,
 // sig[1]
 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f,
 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5e, 0x5d, 0x5f,
 0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6e, 0x6d, 0x6f,
```

```

0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
0x00,
]

```

## NFT Credential

An NFT credential is the same as an [secp256k1 credential](#) with a different TypeID. The TypeID for an NFT credential is `0x0000000e`.

## Unsigned Transactions

Unsigned transactions contain the full content of a transaction with only the signatures missing. Unsigned transactions have four possible types:

[CreateAssetTx](#), [OperationTx](#), [ImportTx](#), and [ExportTx](#). They all embed [BaseTx](#), which contains common fields and operations.

## Unsigned BaseTx

### What Base TX Contains

A base TX contains a `TypeID`, `NetworkID`, `BlockchainID`, `Outputs`, `Inputs`, and `Memo`.

- `TypeID` is the ID for this type. It is `0x00000000`.
- `NetworkID` is an int that defines which network this transaction is meant to be issued to. This value is meant to support transaction routing and is not designed for replay attack prevention.
- `BlockchainID` is a 32-byte array that defines which blockchain this transaction was issued to. This is used for replay attack prevention for transactions that could potentially be valid across network or blockchain.
- `Outputs` is an array of [transferable output objects](#). Outputs must be sorted lexicographically by their serialized representation. The total quantity of the assets created in these outputs must be less than or equal to the total quantity of each asset consumed in the inputs minus the transaction fee.
- `Inputs` is an array of [transferable input objects](#). Inputs must be sorted and unique. Inputs are sorted first lexicographically by their `TxID` and then by the `UTXOIndex` from low to high. If there are inputs that have the same `TxID` and `UTXOIndex`, then the transaction is invalid as this would result in a double spend.
- `Memo` Memo field contains arbitrary bytes, up to 256 bytes.

### Gantt Base TX Specification

|                                               |                                                                     |
|-----------------------------------------------|---------------------------------------------------------------------|
| <code>  type_id : int</code>                  | <code>4 bytes</code>                                                |
| <code>  network_id : int</code>               | <code>4 bytes</code>                                                |
| <code>  blockchain_id : [32]byte</code>       | <code>32 bytes</code>                                               |
| <code>  outputs : []TransferableOutput</code> | <code>4 + size(outputs) bytes</code>                                |
| <code>  inputs : []TransferableInput</code>   | <code>4 + size(inputs) bytes</code>                                 |
| <code>  memo : [256]byte</code>               | <code>4 + size(memo) bytes</code>                                   |
|                                               | <code>  52 + size(outputs) + size(inputs) + size(memo) bytes</code> |

### Proto Base TX Specification

```

message BaseTx {
 uint32 typeID = 1; // 04 bytes
 uint32 network_id = 2; // 04 bytes
 bytes blockchain_id = 3; // 32 bytes
 repeated Output outputs = 4; // 04 bytes + size(outs)
 repeated Input inputs = 5; // 04 bytes + size(ins)
 bytes memo = 6; // 04 bytes + size(memo)
}

```

## Base TX Example

Let's make an base TX that uses the inputs and outputs from the previous examples:

- `TypeID` : 0
- `NetworkID` : 4
- `BlockchainID` : `0xffffffffeeeeeeddddddccccccbbbbbbbaaaaaaaaa9999999988888888`
- `Outputs` :
  - "Example Transferable Output as defined above"
- `Inputs` :
  - "Example Transferable Input as defined above"
- `Memo` : `0x00010203`

```

[
 TypeID <- 0x00000000
 NetworkID <- 0x00000004
 BlockchainID <- 0xffffffffeeeeeddddddccccccbbbbbbaaaaaaa999999988888888
 Outputs <- [
 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0000000700000000000003039000000000000d4310000001000000251025c61;
]
 Inputs <- [
 0xfield1c1b1a19181716151413121101f0e0d0c0b0a0908070605040302010000000005000102030405060708090a0b0c0d0e0f101112131415161718191a1b1
]
 Memo <- 0x00010203
]
=
[
 // typeID
 0x00, 0x00, 0x00, 0x00,
 // networkID:
 0x00, 0x00, 0x00, 0x04,
 // blockchainID:
 0xffff, 0xffff, 0xffff, 0xee, 0xee, 0xee, 0xee,
 0xdd, 0xdd, 0xdd, 0xcc, 0xcc, 0xcc, 0xcc,
 0xbb, 0xbb, 0xbb, 0xaa, 0xaa, 0xaa, 0xaa,
 0x99, 0x99, 0x99, 0x99, 0x88, 0x88, 0x88, 0x88,
 // number of outputs:
 0x00, 0x00, 0x00, 0x01,
 // transferable output:
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
 0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
 0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
 // number of inputs:
 0x00, 0x00, 0x00, 0x01,
 // transferable input:
 0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
 0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
 0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
 0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
 0x00, 0x00, 0x05, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
 0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x00, 0x05,
 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,
 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x07,
 0x00, 0x00, 0x00, 0x03,
 // Memo length:
 0x00, 0x00, 0x00, 0x04,
 // Memo:
 0x00, 0x01, 0x02, 0x03,
]

```

## Unsigned CreateAssetTx

### What Unsigned Create Asset TX Contains

An unsigned create asset TX contains a `BaseTx`, `Name`, `Symbol`, `Denomination`, and `InitialStates`. The `TypeID` is `0x00000001`.

- **BaseTx**
- **Name** is a human readable string that defines the name of the asset this transaction will create. The name is not guaranteed to be unique. The name must consist of only printable ASCII characters and must be no longer than 128 characters.
- **Symbol** is a human readable string that defines the symbol of the asset this transaction will create. The symbol is not guaranteed to be unique. The symbol must consist of only printable ASCII characters and must be no longer than 4 characters.
- **Denomination** is a byte that defines the divisibility of the asset this transaction will create. For example, the AVAX token is divisible into billions. Therefore, the denomination of the AVAX token is 9. The denomination must be no more than 32.
- **InitialStates** is a variable length array that defines the feature extensions this asset supports, and the [initial state](#) of those feature extensions.

## Gantt Unsigned Create Asset TX Specification

```

+-----+-----+
| base_tx : BaseTx | size(base_tx) bytes |
+-----+-----+
| name : string | 2 + len(name) bytes |
+-----+-----+
| symbol : string | 2 + len(symbol) bytes |
+-----+-----+
| denomination : byte | 1 bytes |
+-----+-----+
| initial_states : []InitialState | 4 + size(initial_states) bytes |
+-----+-----+
 | size(base_tx) + size(initial_states) |
 | + 9 + len(name) + len(symbol) bytes |
+-----+-----+

```

Proto Unsigned Create Asset TX Specification

```
message CreateAssetTx {
 BaseTx base_tx = 1; // size(base_tx)
 string name = 2; // 2 bytes + len(name)
 name symbol = 3; // 2 bytes + len(symbol)
 uint8 denomination = 4; // 1 bytes
 repeated InitialState initial_states = 5; // 4 bytes + size(initial_states)
}
```

## Unsigned Create Asset TX Example

Let's make an unsigned base TX that uses the inputs and outputs from the previous examples:

- BaseTx : "Example BaseTx as defined above with ID set to 1"
  - Name : Volatility Index
  - Symbol : VIX
  - Denomination : 2
  - **InitialStates** :
  - "Example Initial State as defined above"

```

0x00, 0x00, 0x00, 0x05, 0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x00, 0x05,
0x00, 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,
0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x07,
0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04,
0x00, 0x01, 0x02, 0x03

// name:
0x00, 0x10, 0x56, 0x6f, 0x6c, 0x61, 0x74, 0x69,
0x6c, 0x69, 0x74, 0x79, 0x20, 0x49, 0x6e, 0x64,
0x65, 0x78,
// symbol length:
0x00, 0x03,
// symbol:
0x56, 0x49, 0x58,
// denomination:
0x02,
// number of InitialStates:
0x00, 0x00, 0x00, 0x01,
// InitialStates[0]:
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
]

```

## Unsigned OperationTx

## What Unsigned Operation TX Contains

An unsigned operation TX contains a `BaseTx`, and `Ops`. The `TypeID` for this type is `0x00000002`.

- **BaseTx**
  - **Ops** is a variable-length array of Transferable Ops.

Gantt Unsigned Operation TX Specification

```

+-----+-----+
| base_tx : BaseTx | size(base_tx) bytes |
+-----+
| ops : []TransferableOp | 4 + size(ops) bytes |
+-----+
| 4 + size(ops) + size(base_tx) bytes |
+-----+

```

Proto Unsigned Operation TX Specification

```
message OperationTx {
 BaseTx base_tx = 1; // size(base_tx)
 repeated TransferOp ops = 2; // 4 bytes + size(ops)
}
```

## Unsigned Operation TX Example

Let's make an unsigned operation TX that uses the inputs and outputs from the previous examples.

- **BaseTx** : "Example BaseTx above" with TypeID set to 2
  - **Ops** : [ "Example Transferable Op as defined above" ]

```
[

// base tx:

0x00, 0x00, 0x00, 0x02,

0x00, 0x00, 0x00, 0x04, 0xff, 0xff, 0xff, 0xff,

0xee, 0xee, 0xee, 0xee, 0xdd, 0xdd, 0xdd, 0xdd,

0xcc, 0xcc, 0xcc, 0xcc, 0xbb, 0xbb, 0xbb, 0xbb,

0xaa, 0xaa, 0xaa, 0xaa, 0x99, 0x99, 0x99, 0x99,

0x88, 0x88, 0x88, 0x88, 0x00, 0x00, 0x00, 0x01,

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,

0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,

0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,

0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,

0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,

0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,

0xde, 0x35, 0x23, 0x2a, 0x4a, 0x46, 0x1c, 0x89,

0x43, 0xab, 0x08, 0x59, 0x00, 0x00, 0x00, 0x01,

0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,

0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,

0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,

0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,

0x00, 0x00, 0x00, 0x05, 0x00, 0x01, 0x02, 0x03,

0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,

0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,

0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,

0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x00, 0x05,

0x00, 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,

0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x07,

0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04,

0x00, 0x01, 0x02, 0x03

// number of operations:

0x00, 0x00, 0x00, 0x01,

// transfer operation:

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

0x08, 0x09, 0xa0, 0xb0, 0xc0, 0xd0, 0xe0, 0xf0,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,

0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,

0x00, 0x00, 0x00, 0x01, 0xf1, 0xe1, 0xd1, 0xc1,

0xb1, 0xa1, 0x91, 0x81, 0x71, 0x61, 0x51, 0x41,

0x31, 0x21, 0x11, 0x01, 0xf0, 0xe0, 0xd0, 0xc0,

0xb0, 0xa0, 0x90, 0x80, 0x70, 0x60, 0x50, 0x40,

0x30, 0x20, 0x10, 0x00, 0x00, 0x00, 0x00, 0x05,

0x00, 0x00, 0x00, 0x0d, 0x00, 0x00, 0x00, 0x00, 0x02,

0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x07,

0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x03,

0x43, 0x11, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00,

0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61, 0xfb,

0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8, 0x34,

0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55, 0xde,

0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e, 0xde,

0x35, 0x23, 0x2a, 0x4a, 0x46, 0x1c, 0x89, 0x43,

0xab, 0x08, 0x59,

]

]
```

## Unsigned ImportTx

### What Unsigned Import TX Contains

An unsigned import TX contains a `BaseTx`, `SourceChain` and `Ins`. \* The TypeID for this type is `0x00000003`.

- `BaseTx`
- `SourceChain` is a 32-byte source blockchain ID.
- `Ins` is a variable length array of [Transferable Inputs](#).

### Gantt Unsigned Import TX Specification

|                                      |                                  |
|--------------------------------------|----------------------------------|
| <code>base_tx : BaseTx</code>        | <code>size(base_tx) bytes</code> |
| <code>source_chain : [32]byte</code> | <code>32 bytes</code>            |
| <code>ins : []TransferIn</code>      | <code>4 + size(ins) bytes</code> |

```
| 36 + size(ins) + size(base_tx) bytes |
```

## Proto Unsigned Import TX Specification

```
message ImportTx {
 BaseTx base_tx = 1; // size(base_tx)
 bytes source_chain = 2; // 32 bytes
 repeated TransferIn ins = 3; // 4 bytes + size(ins)
}
```

## Unsigned Import TX Example

Let's make an unsigned import TX that uses the inputs from the previous examples:

```
0x00, 0x00, 0x00, 0x05,
// assetID:
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
// input:
0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
0x07, 0x5b, 0xcd, 0x15, 0x00, 0x00, 0x00, 0x02,
0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x07,
```

## **Unsigned ExportTx**

## What Unsigned Export TX Contains

An unsigned export TX contains a `BaseTx`, `DestinationChain`, and `Outs`. The `TypeID` for this type is `0x00000004`.

- `DestinationChain` is the 32 byte ID of the chain where the funds are being exported to.
  - `Outs` is a variable length array of [Transferable Outputs](#).

Gantt Unsigned Export TX Specification

```

+-----+-----+
| base_tx : BaseTx | size(base_tx) bytes |
+-----+-----+
| destination_chain : [32]byte | 32 bytes |
+-----+-----+
| outs : []TransferOut | 4 + size(outs) bytes |
+-----+-----+
| | 36 + size(outs) + size(base_tx) bytes |
+-----+-----+

```

## Proto Unsigned Export TX Specification

```
message ExportTx {
 BaseTx base_tx = 1; // size(base_tx)
 bytes destination_chain = 2; // 32 bytes
 repeated TransferOut outs = 3; // 4 bytes + size(outs)
}
```

## Unsigned Export TX Example

Let's make an unsigned export TX that uses the outputs from the previous examples:

```

0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59, 0x00, 0x00, 0x00, 0x01,
0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
0x00, 0x00, 0x00, 0x05, 0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x00, 0x05,
0x00, 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,
0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x07,
0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04,
0x00, 0x01, 0x02, 0x03
// destination_chain:
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// outs[] count:
0x00, 0x00, 0x00, 0x01,
// assetID:
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0xa0, 0xb0, 0xc0, 0xd0, 0xe0, 0xf0,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
// output:
0x00, 0x00, 0x00, 0x07,
0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x39,
0x00, 0x00, 0x00, 0x00, 0x00, 0x44, 0x31,
0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x02,
0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,
0x6d, 0x55, 0xa9, 0x55, 0xc3, 0x34, 0x41, 0x28,
0xe0, 0x60, 0x12, 0x8e, 0xde, 0x35, 0x23, 0xa2,
0x4a, 0x46, 0x1c, 0x89, 0x43, 0xab, 0x08, 0x59,
]

```

## Signed Transaction

A signed transaction is an unsigned transaction with the addition of an array of [credentials](#).

### What Signed Transaction Contains

A signed transaction contains a `CodecID`, `UnsignedTx`, and `Credentials`.

- `CodecID` The only current valid codec id is `00 00`.
- `UnsignedTx` is an unsigned transaction, as described above.
- `Credentials` is an array of [credentials](#). Each credential will be paired with the input in the same index at this credential.

### Gantt Signed Transaction Specification

|                            |  |                                                |
|----------------------------|--|------------------------------------------------|
| -----+-----+-----+         |  |                                                |
| codec_id : uint16          |  | 2 bytes                                        |
| -----+-----+-----+         |  |                                                |
| unsigned_tx : UnsignedTx   |  | size(unsigned_tx) bytes                        |
| -----+-----+-----+         |  |                                                |
| credentials : []Credential |  | 4 + size(credentials) bytes                    |
| -----+-----+-----+         |  |                                                |
|                            |  | 6 + size(unsigned_tx) + len(credentials) bytes |
| -----+-----+-----+         |  |                                                |

### Proto Signed Transaction Specification

```

message Tx {
 uint16 codec_id = 1; // 2 bytes
 UnsignedTx unsigned_tx = 2; // size(unsigned_tx)
 repeated Credential credentials = 3; // 4 bytes + size(credentials)
}

```

### Signed Transaction Example

Let's make a signed transaction that uses the unsigned transaction and credentials from the previous examples.



UTXO

A UTXO is a standalone representation of a transaction output.

## What UTXO Contains

A UTXO contains a `CodecID`, `TxID`, `UTXOIndex`, `AssetID`, and `Output`.

- **CodecID** The only valid `CodecID` is `00 00`
  - **TxID** is a 32-byte transaction ID. Transaction IDs are calculated by taking sha256 of the bytes of the signed transaction.
  - **UTXOIndex** is an int that specifies which output in the transaction specified by `TxID` that this UTXO was created by.
  - **AssetID** is a 32-byte array that defines which asset this UTXO references.
  - **Output** is the output object that created this UTXO. The serialization of Outputs was defined above. Valid output types are [SECP Mint Output](#), [SECP Transfer Output](#), [NFT Mint Output](#), [NFT Transfer Output](#).

## Gantt UTXO Specification

```

+-----+-----+
| codec_id : uint16 | 2 bytes |
+-----+-----+
| tx_id : [32]byte | 32 bytes |
+-----+-----+
| output_index : int | 4 bytes |
+-----+-----+
| asset_id : [32]byte | 32 bytes |
+-----+-----+
| output : Output | size(output) bytes |
+-----+-----+
| | 70 + size(output) bytes |
+-----+-----+

```

## Proto UTXO Specification

```
message Utxo {
 uint16 codec_id = 1; // 02 bytes
 bytes tx_id = 2; // 32 bytes
 uint32 output_index = 3; // 04 bytes
 bytes asset_id = 4; // 32 bytes
 Output output = 5; // size(output)
}
```

## UTXO Examples

Let's make a UTXO with a SECP Mint Output:

```
 0x2e, 0x8c, 0xee, 0x6a, 0x0e, 0xbd, 0x09, 0xf1,
 0xfe, 0x88, 0x4f, 0x68, 0x61, 0xe1, 0xb2, 0x9c,
 0x62, 0x76, 0xaa, 0x2a,
]
```

Let's make a UTXO with a SECP Transfer Output from the signed transaction created above

- **CodecID** : 0
  - **TxID** : 0xf966750f438867c3c9828ddcdbe660e21ccdbb36a9276958f011ba472f75d4e7
  - **UTXOIndex** : 0 = 0x00000000
  - **AssetID** : 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
  - **Output** : "Example SECP256K1 Transferable Output as defined above"

```
[
 CodecID <- 0x0000
 TxID <- 0xf966750f438867c3c9828ddcbe660e21ccdbb36a9276958f011ba472f75d4e7
 UTXOIndex <- 0x00000000
 AssetID <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
 Output <-
0x00000000700000000000003039000000000000d4310000001000000251025c61fbfcfc078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c89c
]
=
[
 // codecID:
 0x00, 0x00,
 // txID:
 0xf9, 0x66, 0x75, 0x0f, 0x43, 0x88, 0x67, 0xc3,
 0xc9, 0x82, 0x8d, 0xdc, 0xdb, 0xe6, 0x60, 0xe2,
 0x1c, 0xcd, 0xbb, 0x36, 0xa9, 0x27, 0x69, 0x58,
 0xf0, 0x11, 0xba, 0x47, 0x2f, 0x75, 0xd4, 0xe7,
 // utxo index:
 0x00, 0x00, 0x00, 0x00,
 // assetID:
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0xa0, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
 // secp transfer output:
 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x02, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
 0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23,
 0x24, 0x25, 0x26, 0x27,
]
```

Let's make a UTXO with an NFT Mint Output:

- **CodeID** : 0
  - **TxID** : 0x03c686ef8d80c519f356929f6da945f7ff90378f0044bb0e1a5d6c1ad06bae7
  - **UTXOIndex** : 0 = 0x00000001
  - **AssetID** : 0x03c686ef8d80c519f356929f6da945f7ff90378f0044bb0e1a5d6c1ad06bae7
  - **Output** : 0x0000000a00000000000000000000000000000000100000013cb7d3842e8ce6a0ebd09f1fe884f6861e1b29c6276aa2a

```
0x03, 0xc6, 0x86, 0xef, 0xe8, 0xd8, 0x0c, 0x51,
0x9f, 0x35, 0x69, 0x29, 0xf6, 0xda, 0x94, 0x5f,
0x7f, 0xe9, 0x03, 0x78, 0xf0, 0x04, 0x4b, 0xb0,
0xel, 0xa5, 0xd6, 0xc1, 0xad, 0x06, 0xba, 0xe7,
// nft mint output:
0x00, 0x00, 0x00, 0xa, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01,
0x3c, 0xb7, 0xd3, 0x84, 0x2e, 0x8c, 0xee, 0x6a,
0x0e, 0xbd, 0x09, 0xf1, 0xfe, 0x88, 0x4f, 0x68,
0x61, 0xel, 0xb2, 0x9c, 0x62, 0x76, 0xaa, 0x2a,
```

Let's make a UTXO with an NFT Transfer Output:

- **CodeID** : 0
  - **TxD** : 0xa68f794a7de7bd5c5db7ba5b73654304731dd586bbf4a6d7b05be6e49de2f936
  - **UTXOIndex** : 0 = 0x00000001
  - **AssetID** : 0x03c686fe8d80c519f356929f6da945f7ff90378f0044bb0e1a5d6c1ad06bae7
  - **Output** :

```
[
 CodecID <- 0x0000
 TxID <- 0xa68f794a7de7bdfc5db7ba5b73654304731dd586bbf4a6d7b05be6e49de2f936
 UTXOIndex <- 0x00000001
 AssetID <- 0x03c686fe8d80c519f356929f6da945f7ff90378f0044bb0e1a5d6clad06bae7
 Output <-
0000000b0000000000000000b4e4654205061796c6f6164000000000000000000000000100000013cb7d3842e8cee6a0ebd09f1fe884f6861elb29c6276aa2a
]
=
[
 // codecID:
 0x00, 0x00,
 // txID:
 0xa6, 0x8f, 0x79, 0x4a, 0x7d, 0xe7, 0xbd, 0xfc,
 0x5d, 0xb7, 0xba, 0x5b, 0x73, 0x65, 0x43, 0x04,
 0x73, 0x1d, 0xd5, 0x86, 0xbb, 0xf4, 0xa6, 0xd7,
 0xb0, 0x5b, 0xe6, 0xe4, 0x9d, 0xe2, 0xf9, 0x36,
 // utxo index:
 0x00, 0x00, 0x00, 0x01,
 // assetID:
 0x03, 0xc6, 0x86, 0xef, 0xe8, 0xd8, 0x0c, 0x51,
 0x9f, 0x35, 0x69, 0x29, 0xf6, 0xda, 0x94, 0x5f,
 0x7f, 0xf9, 0x03, 0x78, 0xf0, 0x04, 0x4b, 0xb0,
 0xe1, 0xa5, 0xd6, 0xc1, 0xad, 0x06, 0xba, 0xe7,
 // nft transfer output:
 0x00, 0x00, 0x00, 0xb, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x00, 0xb, 0x4e, 0x46, 0x54, 0x20,
 0x50, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64, 0x00,
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0x3c,
 0xb7, 0xd3, 0x84, 0x2e, 0x8c, 0xee, 0x6a, 0x0e,
 0xbd, 0x09, 0xf1, 0xfe, 0x88, 0x4f, 0x68, 0x61,
 0xe1, 0xb2, 0x9c, 0x62, 0x76, 0xaa, 0x2a,
]
```

## GenesisAsset

An asset to be issued in an instance of the AVM's Genesis

## What GenesisAsset Contains

An instance of a `GenesisAsset` contains an `Alias`, `NetworkID`, `BlockchainID`, `Outputs`, `Inputs`, `Memo`, `Name`, `Symbol`, `Denomination`, and `InitialStates`.

- **Alias** is the alias for this asset.
  - **NetworkID** defines which network this transaction is meant to be issued to. This value is meant to support transaction routing and is not designed for replay attack prevention.
  - **BlockchainID** is the ID (32-byte array) that defines which blockchain this transaction was issued to. This is used for replay attack prevention for transactions that could potentially be valid across network or blockchain.
  - **Outputs** is an array of [transferable output objects](#). Outputs must be sorted lexicographically by their serialized representation. The total quantity of the assets created in these outputs must be less than or equal to the total quantity of each asset consumed in the inputs minus the transaction fee.
  - **Inputs** is an array of [transferable input objects](#). Inputs must be sorted and unique. Inputs are sorted first lexicographically by their **TxID** and then by the **UTXOIndex** from low to high. If there are inputs that have the same **TxID** and **UTXOIndex**, then the transaction is invalid as this would result in a double spend.

- **Memo** is a memo field that contains arbitrary bytes, up to 256 bytes.
  - **Name** is a human readable string that defines the name of the asset this transaction will create. The name is not guaranteed to be unique. The name must consist of only printable ASCII characters and must be no longer than 128 characters.
  - **Symbol** is a human readable string that defines the symbol of the asset this transaction will create. The symbol is not guaranteed to be unique. The symbol must consist of only printable ASCII characters and must be no longer than 4 characters.
  - **Denomination** is a byte that defines the divisibility of the asset this transaction will create. For example, the AVAX token is divisible into billionths. Therefore, the denomination of the AVAX token is 9. The denomination must be no more than 32.
  - **InitialStates** is a variable length array that defines the feature extensions this asset supports, and the [initial state](#) of those feature extensions.

## Gantt GenesisAsset Specification

```

+-----+-----+
| alias : string | 2 + len(alias) bytes |
+-----+-----+
| network_id : int | 4 bytes |
+-----+-----+
| blockchain_id : [32]bytete | 32 bytes |
+-----+-----+
| outputs : []TransferableOutput | 4 + size(outputs) bytes |
+-----+-----+
| inputs : []TransferableInput | 4 + size(inputs) bytes |
+-----+-----+
| memo : [256]bytete | 4 + size(memo) bytes |
+-----+-----+
| name : string | 2 + len(name) bytes |
+-----+-----+
| symbol : string | 2 + len(symbol) bytes |
+-----+-----+
| denomination : byte | 1 bytes |
+-----+-----+
| initial_states : []InitialState | 4 + size(initial_states) bytes |
+-----+-----+
| 59 + size(alias) + size(outputs) + size(inputs) + size(memo) |
| + len(name) + len(symbol) + size(initial_states) bytes |
+-----+-----+

```

### Proto GenesisAsset Specification

```

```text
message GenesisAsset {
    string alias = 1;                                // 2 bytes + len(alias)
    uint32 network_id = 2;                            // 04 bytes
    bytes blockchain_id = 3;                          // 32 bytes
    repeated Output outputs = 4;                      // 04 bytes + size(outputs)
    repeated Input inputs = 5;                        // 04 bytes + size(inputs)
    bytes memo = 6;                                  // 04 bytes + size(memo)
    string name = 7;                                 // 2 bytes + len(name)
    name symbol = 8;                                // 2 bytes + len(symbol)
    uint8 denomination = 9;                           // 1 bytes
    repeated InitialState initial_states = 10; // 4 bytes + size(initial_states)
}
```

```

## GenesisAsset Example

Let's make a GenesisAsset:

```

Name <- 0x617373657431
Symbol <- 0x66x726f6d20736e6f77666c616b6520746f206176616c616e636865
Denomination <- 0x66x726f6d20736e6f77666c616b6520746f206176616c616e636865
InitialStates <- [
]
]
=
[
 // asset alias len:
 0x00, 0x06,
 // asset alias:
 0x61, 0x73, 0x73, 0x65, 0x74, 0x31,
 // network_id:
 0x00, 0x00, 0x30, 0x39,
 // blockchain_id:
 0x00, 0x00,
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 // output_len:
 0x00, 0x00, 0x00, 0x00,
 // input_len:
 0x00, 0x00, 0x00, 0x00,
 // memo_len:
 0x00, 0x00, 0x00, 0x00,
 // memo:
 0x66, 0x72, 0x6f, 0x6d, 0x20, 0x73, 0x6e, 0x6f, 0x77, 0x66, 0x6c, 0x61,
 0x6b, 0x65, 0x20, 0x74, 0x6f, 0x20, 0x61, 0x76, 0x61, 0x6c, 0x61, 0x6e, 0x63, 0x68, 0x65,
 // asset_name_len:
 0x00, 0x0f,
 // asset_name:
 0x6d, 0x79, 0x46, 0x69, 0x78, 0x65, 0x64, 0x43, 0x61, 0x70, 0x41, 0x73, 0x73, 0x65, 0x74,
 // symbol_len:
 0x00, 0x04,
 // symbol:
 0x4d, 0x46, 0x43, 0x41,
 // denomination:
 0x07,
 // number of InitialStates:
 0x00, 0x00, 0x01,
 // InitialStates[0]:
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
 0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
 0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x55,
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
]

```

Vertex

A vertex is a collection of transactions. It's the DAG equivalent of a block in a linear blockchain.

## What Vertex Contains

A vertex contains a `ChainID`, `Height`, `Epoch`, `ParentIDs`, `TransactionCount`, `TransactionSize`, `Transactions`, and `Restrictions`.

- **ChainID** is the ID of the chain this vertex exists on.
  - **Height** is the maximum height of a parent vertex plus 1.
  - **Epoch** is the epoch this vertex belongs to.
  - **ParentIDs** are the IDs of this vertex's parents.
  - **TransactionCount** is the total number of transactions in this vertex.
  - **TransactionSize** is the total size of the transactions in this vertex.
  - **Transactions** are the transactions in this vertex.
  - **Restrictions** are IDs of transactions that must be accepted in the same epoch as this vertex or an earlier one.

## Gantt Vertex Specification

|          |              |          |       |
|----------|--------------|----------|-------|
| -----    | -----        | -----    |       |
| chain_id | : [32]bytete | 32 bytes |       |
| -----    | -----        | -----    | ----- |
| height   | : long       | 8 bytes  |       |
| -----    | -----        | -----    | ----- |

```

+-----+-----+
| epoch : int | 4 bytes |
+-----+-----+
| parent_ids : []ParentID | 4 + size(parent_ids) bytes |
+-----+-----+
| tx_count : int | 4 bytes |
+-----+-----+
| txs_size : int | 4 bytes |
+-----+-----+
| transactions : []Transaction | 4 + size(transactions) bytes |
+-----+-----+
| restrictions : []Restriction | 4 + size(restrictions) bytes |
+-----+-----+
| 64 + size(parentIDs) + size(restrictions) + size(transactions) bytes |
+-----+-----+

```

### Proto Vertex Specification

```

message Vertex {
 uint16 codec_id = 1; // 04 bytes
 bytes chain_id = 2; // 32 bytes
 uint64 height = 3; // 08 bytes
 uint32 epoch = 4; // 04 bytes
 repeated bytes parent_ids = 5; // 04 bytes + 32 bytes * count(parent_ids)
 uint32 tx_count = 5; // 04 bytes
 uint32 txs_size = 6; // 04 bytes
 repeated bytes transactions = 9; // 04 bytes + size(transactions)
 repeated bytes restrictions = 10; // 04 bytes + size(restrictions)
}

```

### Vertex Example

Let's make a Vertex:

- **CodecID** : 0x0000
- **ChainID** : 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
- **Height** : 3
- **Epoch** : 0
- **ParentIDs** : ["0x73fa32c486fe9feeb392ee374530c6fe076b08a111fd58e974e7f903a52951d2"]
- **Transactions** : [Example BaseTx as defined above]
- **Restrictions** : []

```

[
 CodecID <- 0x0000
 ChainID <- 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
 Height <- 0x0000000000000003
 Epoch <- 0x00000000
 ParentIDs <- [0x73fa32c486fe9feeb392ee374530c6fe076b08a111fd58e974e7f903a52951d2]
 Transactions <- [Example BaseTx defined above]
 Restrictions <- []
]
=
[
 // codec id
 00 00
 // chain id
 d8 91 ad 56 05 6d 9c 01 f1 8f 43 f5 8b 5c 78 4a d0 7a 4a 49 cf 3d 1f 11 62 38 04 b5 cb a2 c6 bf
 // height
 00 00 00 00 00 00 00 03
 // epoch
 00 00 00 00
 // num parent IDs
 00 00 00 01
 // parent id 1
 73 fa 32 c4 86 fe 9f ee b3 92 ee 37 45 30 c6 fe 07 6b 08 a1 11 fd 58 e9 74 e7 f9 03 a5 29 51 d2
 // num txs
 00 00 00 01
 // size of the transaction
 00 00 01 7b
 // base tx from above
 [omitted for brevity]
 // num restrictions
 00 00 00 00
]

```

## Banff Changes

This document specifies the changes in Avalanche "Banff", which will be released in AvalancheGo v1.9.x.

### Block Changes

#### Apricot

Apricot allows the following block types with the following content:

- *Standard Blocks* may contain multiple transactions of the following types:
  - CreateChainTx
  - CreateSubnetTx
  - ImportTx
  - ExportTx
- *Proposal Blocks* may contain a single transaction of the following types:
  - AddValidatorTx
  - AddDelegatorTx
  - AddSubnetValidatorTx
  - RewardValidatorTx
  - AdvanceTimeTx
- *Options Blocks*, that is *Commit Block* and *Abort Block* do not contain any transactions.

Each block has a header containing:

- ParentID
- Height

#### Banff

Banff allows the following block types with the following content:

- *Standard Blocks* may contain multiple transactions of the following types:
  - CreateChainTx
  - CreateSubnetTx
  - ImportTx
  - ExportTx
  - AddValidatorTx
  - AddDelegatorTx
  - AddSubnetValidatorTx
  - RemoveSubnetValidatorTx
  - TransformSubnetTx
  - AddPermissionlessValidatorTx
  - AddPermissionlessDelegatorTx
- *Proposal Blocks* may contain a single transaction of the following types:
  - RewardValidatorTx
- *Options blocks*, that is *Commit Block* and *Abort Block* do not contain any transactions.

Note that each block has an header containing:

- ParentID
- Height
- Time

So the two main differences with respect to Apricot are:

- *AddValidatorTx*, *AddDelegatorTx*, *AddSubnetValidatorTx* are included into Standard Blocks rather than Proposal Blocks so that they don't need to be voted on (that is followed by a Commit/Abort Block).
- New Transaction types: *RemoveSubnetValidatorTx*, *TransformSubnetTx*, *AddPermissionlessValidatorTx*, and *AddPermissionlessDelegatorTx* have been added into Standard Blocks.
- Block timestamp is explicitly serialized into block header, to allow chain time update.

### New Transactions

#### RemoveSubnetValidatorTx

```
type RemoveSubnetValidatorTx struct {
 BaseTx `serialize:"true"`
 // The node to remove from the subnet.
 NodeID ids.NodeID `serialize:"true" json:"nodeID"`
 // The subnet to remove the node from.
 Subnet ids.ID `serialize:"true" json:"subnet"`
 // Proves that the issuer has the right to remove the node from the subnet.
 SubnetAuth verify.Verifiable `serialize:"true" json:"subnetAuthorization"`
}
```

### TransformSubnetTx

```
type TransformSubnetTx struct {
 // Metadata, inputs and outputs
 BaseTx `serialize:"true"`
 // ID of the Subnet to transform
 // Restrictions:
 // - Must not be the Primary Network ID
 Subnet ids.ID `serialize:"true" json:"subnetID"`
 // Asset to use when staking on the Subnet
 // Restrictions:
 // - Must not be the Empty ID
 // - Must not be the AVAX ID
 AssetID ids.ID `serialize:"true" json:"assetID"`
 // Amount to initially specify as the current supply
 // Restrictions:
 // - Must be > 0
 InitialSupply uint64 `serialize:"true" json:"initialSupply"`
 // Amount to specify as the maximum token supply
 // Restrictions:
 // - Must be >= [InitialSupply]
 MaximumSupply uint64 `serialize:"true" json:"maximumSupply"`
 // MinConsumptionRate is the rate to allocate funds if the validator's stake
 // duration is 0
 MinConsumptionRate uint64 `serialize:"true" json:"minConsumptionRate"`
 // MaxConsumptionRate is the rate to allocate funds if the validator's stake
 // duration is equal to the minting period
 // Restrictions:
 // - Must be >= [MinConsumptionRate]
 // - Must be <= [reward.PercentDenominator]
 MaxConsumptionRate uint64 `serialize:"true" json:"maxConsumptionRate"`
 // MinValidatorStake is the minimum amount of funds required to become a
 // validator.
 // Restrictions:
 // - Must be > 0
 // - Must be <= [InitialSupply]
 MinValidatorStake uint64 `serialize:"true" json:"minValidatorStake"`
 // MaxValidatorStake is the maximum amount of funds a single validator can
 // be allocated, including delegated funds.
 // Restrictions:
 // - Must be >= [MinValidatorStake]
 // - Must be <= [MaximumSupply]
 MaxValidatorStake uint64 `serialize:"true" json:"maxValidatorStake"`
 // MinStakeDuration is the minimum number of seconds a staker can stake for.
 // Restrictions:
 // - Must be > 0
 MinStakeDuration uint32 `serialize:"true" json:"minStakeDuration"`
 // MaxStakeDuration is the maximum number of seconds a staker can stake for.
 // Restrictions:
 // - Must be >= [MinStakeDuration]
 // - Must be <= [GlobalMaxStakeDuration]
 MaxStakeDuration uint32 `serialize:"true" json:"maxStakeDuration"`
 // MinDelegationFee is the minimum percentage a validator must charge a
 // delegator for delegating.
 // Restrictions:
 // - Must be <= [reward.PercentDenominator]
 MinDelegationFee uint32 `serialize:"true" json:"minDelegationFee"`
 // MinDelegatorStake is the minimum amount of funds required to become a
 // delegator.
 // Restrictions:
 // - Must be > 0
 MinDelegatorStake uint64 `serialize:"true" json:"minDelegatorStake"`
 // MaxValidatorWeightFactor is the factor which calculates the maximum
 // amount of delegation a validator can receive.
 // Note: a value of 1 effectively disables delegation.
 // Restrictions:
 // - Must be > 0
 MaxValidatorWeightFactor byte `serialize:"true" json:"maxValidatorWeightFactor"`
 // UptimeRequirement is the minimum percentage a validator must be online
 // and responsive to receive a reward.
 // Restrictions:
 // - Must be <= [reward.PercentDenominator]
 UptimeRequirement uint32 `serialize:"true" json:"uptimeRequirement"`
 // Authorizes this transformation
 SubnetAuth verify.Verifiable `serialize:"true" json:"subnetAuthorization"`
}
```

#### AddPermissionlessValidatorTx

```
type AddPermissionlessValidatorTx struct {
 // Metadata, inputs and outputs
 BaseTx `serialize:"true"`
 // Describes the validator
 Validator validator.Validator `serialize:"true" json:"validator"`
 // ID of the subnet this validator is validating
 Subnet ids.ID `serialize:"true" json:"subnet"`
 // Where to send staked tokens when done validating
 StakeOuts []*avax.TransferableOutput `serialize:"true" json:"stake"`
 // Where to send validation rewards when done validating
 ValidatorRewardsOwner fx.Owner `serialize:"true" json:"validationRewardsOwner"`
 // Where to send delegation rewards when done validating
 DelegatorRewardsOwner fx.Owner `serialize:"true" json:"delegationRewardsOwner"`
 // Fee this validator charges delegators as a percentage, times 10,000
 // For example, if this validator has DelegationShares=300,000 then they
 // take 30% of rewards from delegators
 DelegationShares uint32 `serialize:"true" json:"shares"`
}
```

#### AddPermissionlessDelegatorTx

```
type AddPermissionlessDelegatorTx struct {
 // Metadata, inputs and outputs
 BaseTx `serialize:"true"`
 // Describes the validator
 Validator validator.Validator `serialize:"true" json:"validator"`
 // ID of the subnet this validator is validating
 Subnet ids.ID `serialize:"true" json:"subnet"`
 // Where to send staked tokens when done validating
 Stake []*avax.TransferableOutput `serialize:"true" json:"stake"`
 // Where to send staking rewards when done validating
 RewardsOwner fx.Owner `serialize:"true" json:"rewardsOwner"`
}
```

#### New TypeIDs

```
ApricotProposalBlock = 0
ApricotAbortBlock = 1
ApricotCommitBlock = 2
ApricotStandardBlock = 3
ApricotAtomicBlock = 4

secp256klfx.TransferInput = 5
secp256klfx.MintOutput = 6
secp256klfx.TransferOutput = 7
secp256klfx.MintOperation = 8
secp256klfx.Credential = 9
secp256klfx.Input = 10
secp256klfx.OutputOwners = 11

AddValidatorTx = 12
AddSubnetValidatorTx = 13
AddDelegatorTx = 14
CreateChainTx = 15
CreateSubnetTx = 16
ImportTx = 17
ExportTx = 18
AdvanceTimeTx = 19
RewardValidatorTx = 20

stakeable.LockIn = 21
stakeable.LockOut = 22

RemoveSubnetValidatorTx = 23
TransformSubnetTx = 24
AddPermissionlessValidatorTx = 25
AddPermissionlessDelegatorTx = 26

EmptyProofOfPossession = 27
BLSProofOfPossession = 28

BanffProposalBlock = 29
BanffAbortBlock = 30
```

```
BanffCommitBlock = 31
BanffStandardBlock = 32
```

## Coreth Atomic Transaction Format

This page is meant to be the single source of truth for how we serialize atomic transactions in `Coreth`. This document uses the [primitive serialization](#) format for packing and [secp256k1](#) for cryptographic user identification.

### Codec ID

Some data is prepended with a codec ID (uint16) that denotes how the data should be deserialized. Right now, the only valid codec ID is 0 ( 0x00 0x00 ).

### Inputs

Inputs to Coreth Atomic Transactions are either an `EVMInput` from this chain or a `TransferableInput` (which contains a `SECP256K1TransferInput`) from another chain. The `EVMInput` will be used in `ExportTx` to spend funds from this chain, while the `TransferableInput` will be used to import atomic UTXOs from another chain.

#### EVM Input

Input type that specifies an EVM account to deduct the funds from as part of an `ExportTx`.

##### What EVM Input Contains

An EVM Input contains an `address`, `amount`, `assetID`, and `nonce`.

- `Address` is the EVM address from which to transfer funds.
- `Amount` is the amount of the asset to be transferred (specified in nAVAX for AVAX and the smallest denomination for all other assets).
- `AssetID` is the ID of the asset to transfer.
- `Nonce` is the nonce of the EVM account exporting funds.

##### Gantt EVM Input Specification

|                     |          |  |
|---------------------|----------|--|
| +-----+-----+-----+ |          |  |
| address : [20]byte  | 20 bytes |  |
| +-----+-----+-----+ |          |  |
| amount : uint64     | 08 bytes |  |
| +-----+-----+-----+ |          |  |
| asset_id : [32]byte | 32 bytes |  |
| +-----+-----+-----+ |          |  |
| nonce : uint64      | 08 bytes |  |
| +-----+-----+-----+ |          |  |
| 68 bytes            |          |  |
| +-----+-----+-----+ |          |  |

##### Proto EVM Input Specification

```
message {
 bytes address = 1; // 20 bytes
 uint64 amount = 2; // 08 bytes
 bytes assetID = 3; // 32 bytes
 uint64 nonce = 4; // 08 bytes
}
```

##### EVM Input Example

Let's make an EVM Input:

- `Address`: 0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc
- `Amount`: 2000000
- `AssetID`: 0x000102030405060708090a0b0c0d0e0f101112131415161718191alblclidelf
- `Nonce`: 0

```
[
 Address <- 0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc,
 Amount <- 0x000000000001e8480
 AssetID <- 0xdbc890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
 Nonce <- 0x0000000000000000
]
=
[
 // address:
 0x8d, 0xb9, 0x7c, 0x7c, 0xec, 0xe2, 0x49, 0xc2,
 0xb9, 0x8b, 0xdc, 0x02, 0x26, 0xcc, 0x4c, 0x2a,
```

```

0x57, 0xbff, 0x52, 0xfc,
// amount:
0x00, 0x00, 0x00, 0x00, 0x00, 0x1e, 0x84, 0x80,
// assetID:
0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
// nonce:
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
]

```

## Transferable Input

Transferable Input wraps a `SECP256K1TransferInput`. Transferable inputs describe a specific UTXO with a provided transfer input.

### What Transferable Input Contains

A transferable input contains a `TxID`, `UTXOIndex` `AssetID` and an `Input`.

- `TxID` is a 32-byte array that defines which transaction this input is consuming an output from.
- `UTXOIndex` is an int that defines which utxo this input is consuming in the specified transaction.
- `AssetID` is a 32-byte array that defines which asset this input references.
- `Input` is a `SECP256K1TransferInput`, as defined below.

### Gantt Transferable Input Specification

|                                      |                                         |
|--------------------------------------|-----------------------------------------|
| <code>  tx_id : [32]byte  </code>    | <code>32 bytes  </code>                 |
| <code>  utxo_index : int  </code>    | <code>04 bytes  </code>                 |
| <code>  asset_id : [32]byte  </code> | <code>32 bytes  </code>                 |
| <code>  input : Input  </code>       | <code>size(input) bytes  </code>        |
|                                      | <code>  68 + size(input) bytes  </code> |

### Proto Transferable Input Specification

```

message TransferableInput {
 bytes tx_id = 1; // 32 bytes
 uint32 utxo_index = 2; // 04 bytes
 bytes asset_id = 3; // 32 bytes
 Input input = 4; // size(input)
}

```

### Transferable Input Example

Let's make a transferable input:

- `TxID`: `0x6613a40dcdd8d22ea4aa99a4c84349056317cf550b6685e045e459954f258e59`
- `UTXOIndex`: `1`
- `AssetID`: `0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db`
- `Input`: "Example SECP256K1 Transfer Input from below"

```

[
 TxID <- 0x6613a40dcdd8d22ea4aa99a4c84349056317cf550b6685e045e459954f258e59
 UTXOIndex <- 0x00000001
 AssetID <- 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
 Input <- 0x0000000500000000075bcd15000000200000070000003
]
=
[
 // txID:
 0x66, 0x13, 0xa4, 0x0d, 0xcd, 0xd8, 0xd2, 0x2e,
 0xa4, 0xaa, 0x99, 0xa4, 0xc8, 0x43, 0x49, 0x05,
 0x63, 0x17, 0xcf, 0x55, 0x0b, 0x66, 0x85, 0xe0,
 0x45, 0xe4, 0x59, 0x95, 0x4f, 0x25, 0x8e, 0x59,
 // utxoIndex:
 0x00, 0x00, 0x00, 0x01,
 // assetID:
 0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
 0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
 // input:
 0x0000000500000000075bcd15000000200000070000003
]

```

```

0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
// input:
0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x74,
0xa, 0x52, 0x88, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00,
]

```

## SECP256K1 Transfer Input

A [secp256k1](#) transfer input allows for spending an unspent secp256k1 transfer output.

### What SECP256K1 Transfer Input Contains

A secp256k1 transfer input contains an `Amount` and `AddressIndices`.

- `TypeID` is the ID for this input type. It is `0x00000005`.
- `Amount` is a long that specifies the quantity that this input should be consuming from the UTXO. Must be positive. Must be equal to the amount specified in the UTXO.
- `AddressIndices` is a list of unique ints that define the private keys that are being used to spend the UTXO. Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.

### Gantt SECP256K1 Transfer Input Specification

|                                          |                                                      |
|------------------------------------------|------------------------------------------------------|
| <code>  type_id : int  </code>           | <code>4 bytes  </code>                               |
| <code>  amount : long  </code>           | <code>8 bytes  </code>                               |
| <code>  address_indices : []int  </code> | <code>4 + 4 * len(address_indices) bytes  </code>    |
|                                          | <code>  16 + 4 * len(address_indices) bytes  </code> |

### Proto SECP256K1 Transfer Input Specification

```

message SECP256K1TransferInput {
 uint32 typeID = 1; // 04 bytes
 uint64 amount = 2; // 08 bytes
 repeated uint32 address_indices = 3; // 04 bytes + 04 bytes * len(address_indices)
}

```

### SECP256K1 Transfer Input Example

Let's make a payment input with:

- `TypeID` : 5
- `Amount` : 5000000000000000
- `AddressIndices` : [0]

```

[
 TypeID <- 0x00000005
 Amount <- 5000000000000000 = 0x000000746a528800,
 AddressIndices <- [0x00000000]
]
=
[
 // type id:
 0x00, 0x00, 0x00, 0x05,
 // amount:
 0x00, 0x00, 0x00, 0x74, 0x6a, 0x52, 0x88, 0x00,
 // length:
 0x00, 0x00, 0x00, 0x01,
 // sig[0]
 0x00, 0x00, 0x00, 0x00,
]

```

## Outputs

Outputs to Coreth Atomic Transactions are either an `EVMOutput` to be added to the balance of an address on this chain or a `TransferableOutput` (which contains a `SECP256K1TransferOutput`) to be moved to another chain.

The EVM Output will be used in `ImportTx` to add funds to this chain, while the `TransferableOutput` will be used to export atomic UTXOs to another chain.

## EVM Output

Output type specifying a state change to be applied to an EVM account as part of an `ImportTx`.

### What EVM Output Contains

An EVM Output contains an `address`, `amount`, and `assetID`.

- `Address` is the EVM address that will receive the funds.
- `Amount` is the amount of the asset to be transferred (specified in nAVAX for AVAX and the smallest denomination for all other assets).
- `AssetID` is the ID of the asset to transfer.

### Gantt EVM Output Specification

|                                |
|--------------------------------|
| +-----+-----+-----+            |
| address : [20]byte   20 bytes  |
| +-----+-----+-----+            |
| amount : uint64   08 bytes     |
| +-----+-----+-----+            |
| asset_id : [32]byte   32 bytes |
| +-----+-----+-----+            |
| 60 bytes                       |
| +-----+-----+-----+            |

### Proto EVM Output Specification

```
message {
 bytes address = 1; // 20 bytes
 uint64 amount = 2; // 08 bytes
 bytes assetID = 3; // 32 bytes
}
```

### EVM Output Example

Let's make an EVM Output:

- Address: 0x0eb5ccb85c29009b6060decb353a38ea3b52cd20
- Amount: 500000000000
- AssetID: 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db

```
[
 Address <- 0x0eb5ccb85c29009b6060decb353a38ea3b52cd20,
 Amount <- 0x000000746a528800
 AssetID <- 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
]
=
[
 // address:
 0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
 0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
 0x43, 0xab, 0x08, 0x59,
 // amount:
 0x00, 0x00, 0x00, 0x74, 0x6a, 0x52, 0x88, 0x00,
 // assetID:
 0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
 0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
 0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
 0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
]
```

## Transferable Output

Transferable outputs wrap a `SECP256K1TransferOutput` with an asset ID.

### What Transferable Output Contains

A transferable output contains an `AssetID` and an `Output` which is a `SECP256K1TransferOutput`.

- `AssetID` is a 32-byte array that defines which asset this output references.
- `Output` is a `SECP256K1TransferOutput` as defined below.

### Gantt Transferable Output Specification

|                                |
|--------------------------------|
| +-----+-----+-----+            |
| asset_id : [32]byte   32 bytes |
| +-----+-----+-----+            |

|                                  |                                   |
|----------------------------------|-----------------------------------|
| <code>  output : Output  </code> | <code>size(output) bytes  </code> |
| <code>+-----+-----+</code>       | <code>+-----+-----+</code>        |
| <code>  32 +</code>              | <code>size(output) bytes  </code> |
| <code>+-----+</code>             | <code>+-----+</code>              |

### Proto Transferable Output Specification

```
message TransferableOutput {
 bytes asset_id = 1; // 32 bytes
 Output output = 2; // size(output)
}
```

### Transferable Output Example

Let's make a transferable output:

- AssetID: 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
- Output: "Example SECP256K1 Transfer Output from below"

```
[
 AssetID <- 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
 Output <-
0x00000007000000000000303900000000000d43100000010000000251025c61fbfc078f6934f834be6dd26d55a955c3344128e060128ede3523a24a461c894
]
=
[
 // assetID:
 0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
 0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
 0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
 0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
 // output:
 0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
 0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
 0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
 0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x0f, 0x42, 0x40, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
 0x00, 0x00, 0x00, 0x01, 0x66, 0xf9, 0xd, 0xb6,
 0x13, 0x7a, 0x78, 0xf7, 0x6b, 0x36, 0x93, 0xf7,
 0xf2, 0xbc, 0x50, 0x79, 0x56, 0xda, 0xe5, 0x63,
]
```

### SECP256K1 Transfer Output

A [secp256k1](#) transfer output allows for sending a quantity of an asset to a collection of addresses after a specified Unix time.

#### What SECP256K1 Transfer Output Contains

A secp256k1 transfer output contains a `TypeID`, `Amount`, `Locktime`, `Threshold`, and `Addresses`.

- `TypeID` is the ID for this output type. It is `0x00000007`.
- `Amount` is a long that specifies the quantity of the asset that this output owns. Must be positive.
- `Locktime` is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- `Threshold` is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`. If `Addresses` is empty, must be 0.
- `Addresses` is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

#### Gantt SECP256K1 Transfer Output Specification

|                                           |                                              |
|-------------------------------------------|----------------------------------------------|
| <code>  type_id : int  </code>            | <code>4 bytes  </code>                       |
| <code>+-----+-----+</code>                | <code>+-----+-----+</code>                   |
| <code>  amount : long  </code>            | <code>8 bytes  </code>                       |
| <code>+-----+-----+</code>                | <code>+-----+-----+</code>                   |
| <code>  locktime : long  </code>          | <code>8 bytes  </code>                       |
| <code>+-----+-----+</code>                | <code>+-----+-----+</code>                   |
| <code>  threshold : int  </code>          | <code>4 bytes  </code>                       |
| <code>+-----+-----+</code>                | <code>+-----+-----+</code>                   |
| <code>  addresses : [][20]bytete  </code> | <code>4 + 20 * len(addresses) bytes  </code> |
| <code>+-----+-----+</code>                | <code>+-----+-----+</code>                   |

```
| 28 + 20 * len(addresses) bytes |
+-----+
```

## Proto SECP256K1 Transfer Output Specification

```
message SECP256K1TransferOutput {
 uint32 typeID = 1; // 04 bytes
 uint64 amount = 2; // 08 bytes
 uint64 locktime = 3; // 08 bytes
 uint32 threshold = 4; // 04 bytes
 repeated bytes addresses = 5; // 04 bytes + 20 bytes * len(addresses)
}
```

## SECP256K1 Transfer Output Example

Let's make a secp256k1 transfer output with:

- **TypeID** : 7
- **Amount** : 1000000
- **Locktime** : 0
- **Threshold** : 1
- **Addresses** :
  - 0x66f90db6137a78f76b3693f7f2bc507956dae563

```
[
 TypeID <- 0x00000007
 Amount <- 0x000000000000f4240
 Locktime <- 0x0000000000000000
 Threshold <- 0x00000001
 Addresses <- [
 0x66f90db6137a78f76b3693f7f2bc507956dae563
]
]
=
[
 // typeID:
 0x00, 0x00, 0x00, 0x07,
 // amount:
 0x00, 0x00, 0x00, 0x00, 0x00, 0x0f, 0x42, 0x40,
 // locktime:
 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
 // threshold:
 0x00, 0x00, 0x00, 0x01,
 // number of addresses:
 0x00, 0x00, 0x00, 0x01,
 // addrs[0]:
 0x66, 0xf9, 0x0d, 0xb6, 0x13, 0x7a, 0x78, 0xf7,
 0x6b, 0x36, 0x93, 0xf7, 0xf2, 0xbc, 0x50, 0x79,
 0x56, 0xda, 0xe5, 0x63,
]
```

## Atomic Transactions

Atomic Transactions are used to move funds between chains. There are two types `ImportTx` and `ExportTx`.

### ExportTx

`ExportTx` is a transaction to export funds from Coreth to a different chain.

#### What ExportTx Contains

An `ExportTx` contains an `typeID`, `networkID`, `blockchainID`, `destinationChain`, `inputs`, and `exportedOutputs`.

- **typeID** is an int that the type for an `ExportTx`. The `typeID` for an `exportTx` is 1.
- **networkID** is an int that defines which Avalanche network this transaction is meant to be issued to. This could refer to Mainnet, Fuji, etc. and is different than the EVM's network ID.
- **blockchainID** is a 32-byte array that defines which blockchain this transaction was issued to.
- **destinationChain** is a 32-byte array that defines which blockchain this transaction exports funds to.
- **inputs** is an array of EVM Inputs to fund the `ExportTx`.
- **exportedOutputs** is an array of `TransferableOutputs` to be transferred to `destinationChain`.

#### Gantt ExportTx Specification

|                                 |                                     |                                                                  |
|---------------------------------|-------------------------------------|------------------------------------------------------------------|
| <code>  typeID</code>           | <code>: int</code>                  | <code>  04 bytes  </code>                                        |
| <code>  networkID</code>        | <code>: int</code>                  | <code>  04 bytes  </code>                                        |
| <code>  blockchainID</code>     | <code>: [32]byte</code>             | <code>  32 bytes  </code>                                        |
| <code>  destinationChain</code> | <code>: [32]byte</code>             | <code>  32 bytes  </code>                                        |
| <code>  inputs</code>           | <code>: []EvmInput</code>           | <code>  4 + size(inputs) bytes  </code>                          |
| <code>  exportedOutputs</code>  | <code>: []TransferableOutput</code> | <code>  4 + size(exportedOutputs) bytes  </code>                 |
|                                 |                                     | <code>  80 + size(inputs) + size(exportedOutputs) bytes  </code> |

### ExportTx Example

Let's make an EVM Output:

- `TypeID` : 1
- `NetworkID` : 12345
- `BlockchainID` : 0x91060eabfb5a571720109b5896e5ff00010a1cf6b103d585e6ebf27b97a1735
- `DestinationChain` : 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
- `Inputs` :
  - "Example EVMInput as defined above"
- `Exportedoutputs` :
  - "Example TransferableOutput as defined above"

```
[
 TypeID <- 0x00000001
 NetworkID <- 0x00003039
 BlockchainID <- 0x91060eabfb5a571720109b5896e5ff00010a1cf6b103d585e6ebf27b97a1735
 DestinationChain <- 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
 Inputs <- [
 0xc3344128e060128ede3523a24a461c8943ab08590000000000003039000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f00000000(
]
 ExportedOutputs <- [
 0xdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2dbdbcf890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db(
]
]
 =
 [
 // typeID:
 0x00, 0x00, 0x00, 0x01,
 // networkID:
 0x00, 0x00, 0x00, 0x04,
 // blockchainID:
 0x91, 0x06, 0x0e, 0xab, 0xfb, 0x5a, 0x57, 0x17,
 0x20, 0x10, 0x9b, 0x58, 0x96, 0xe5, 0xff, 0x00,
 0x01, 0xa0, 0x1c, 0xfe, 0x6b, 0x10, 0x3d, 0x58,
 0x5e, 0x6e, 0xbf, 0x27, 0xb9, 0x7a, 0x17, 0x35,
 // destination_chain:
 0xd8, 0x91, 0xad, 0x56, 0x05, 0x6d, 0x9c, 0x01,
 0xf1, 0x8f, 0x43, 0x5f, 0x8b, 0x5c, 0x78, 0x4a,
 0xd0, 0x7a, 0x4a, 0x49, 0xcf, 0x3d, 0x1f, 0x11,
 0x62, 0x38, 0x04, 0xb5, 0xcb, 0xa2, 0xc6, 0xbf,
 // inputs[] count:
 0x00, 0x00, 0x00, 0x01,
 // inputs[0]
 0x8d, 0xb9, 0x7c, 0x7c, 0xec, 0xe2, 0x49, 0xc2,
 0xb9, 0x8b, 0xdc, 0x02, 0x26, 0xcc, 0x4c, 0x2a,
 0x57, 0xbf, 0x52, 0xfc, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x1e, 0x84, 0x80, 0xdb, 0xcf, 0x89, 0x0f,
 0x77, 0xf4, 0x9b, 0x96, 0x85, 0x76, 0x48, 0xb7,
 0x2b, 0x77, 0xf9, 0xf8, 0x29, 0x37, 0xf2, 0x8a,
 0x68, 0x70, 0x4a, 0xf0, 0x5d, 0xa0, 0xdc, 0x12,
 0xba, 0x53, 0xf2, 0xdb, 0x00, 0x00, 0x00, 0x00,
 0x00, 0x00, 0x00, 0x00,
 // exportedOutputs[] count:
 0x00, 0x00, 0x01,
 // exportedOutputs[0]
 0xdb, 0xcf, 0x89, 0x0f, 0x77, 0xf4, 0x9b, 0x96,
```

```

0x85, 0x76, 0x48, 0xb7, 0x2b, 0x77, 0xf9, 0xf8,
0x29, 0x37, 0xf2, 0x8a, 0x68, 0x70, 0x4a, 0xf0,
0x5d, 0xa0, 0xdc, 0x12, 0xba, 0x53, 0xf2, 0xdb,
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
0x00, 0x0f, 0x42, 0x40, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x01, 0x66, 0xf9, 0xd, 0xb6,
0x13, 0x7a, 0x78, 0xf7, 0x6b, 0x36, 0x93, 0xf7,
0xf2, 0xbc, 0x50, 0x79, 0x56, 0xda, 0xe5, 0x63,
]

```

## ImportTx

ImportTx is a transaction to import funds to Coreth from another chain.

### What ImportTx Contains

An ImportTx contains an `typeID`, `networkID`, `blockchainID`, `destinationChain`, `importedInputs`, and `Outs`.

- `typeID` is an int that the type for an ImportTx. The `typeID` for an `ImportTx` is 0.
- `networkID` is an int that defines which Avalanche network this transaction is meant to be issued to. This could refer to Mainnet, Fuji, etc. and is different than the EVM's network ID.
- `blockchainID` is a 32-byte array that defines which blockchain this transaction was issued to.
- `sourceChain` is a 32-byte array that defines which blockchain from which to import funds.
- `importedInputs` is an array of TransferableInputs to fund the ImportTx.
- `Outs` is an array of EVM Outputs to be imported to this chain.

### Gantt ImportTx Specification

|                                                   |                                              |                                |
|---------------------------------------------------|----------------------------------------------|--------------------------------|
| +-----+-----+                                     |                                              | +-----+                        |
| <code>typeID</code> : int                         |                                              | 04 bytes                       |
| +-----+-----+                                     |                                              | +-----+                        |
| <code>networkID</code> : int                      |                                              | 04 bytes                       |
| +-----+-----+                                     |                                              | +-----+                        |
| <code>blockchainID</code> : [32]byte              |                                              | 32 bytes                       |
| +-----+-----+                                     |                                              | +-----+                        |
| <code>sourceChain</code> : [32]byte               |                                              | 32 bytes                       |
| +-----+-----+                                     |                                              | +-----+                        |
| <code>importedInputs</code> : []TransferableInput |                                              | 4 + size(importedInputs) bytes |
| +-----+-----+                                     |                                              | +-----+                        |
| <code>outs</code> : []EVMOutput                   |                                              | 4 + size(outs) bytes           |
| +-----+-----+                                     |                                              | +-----+                        |
|                                                   | 80 + size(importedInputs) + size(outs) bytes |                                |
| +-----+-----+                                     |                                              | +-----+                        |

### ImportTx Example

Let's make an ImportTx:

- `TypeID` : 0
- `NetworkID` : 12345
- `BlockchainID` : 0x91060eabfb5a571720109b5896e5ff00010a1cf6b103d585e6ebf27b97a1735
- `SourceChain` : 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
- `ImportedInputs` :
  - "Example TransferableInput as defined above"
- `Outs` :
  - "Example EVMOutput as defined above"

```
[
 TypeID <- 0x00000000
 NetworkID <- 0x00003039
 BlockchainID <- 0x91060eabfb5a571720109b5896e5ff00010a1cf6b103d585e6ebf27b97a1735
 SourceChain <- 0xd891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1f11623804b5cba2c6bf
 ImportedInputs <- [
 0x6613a40dcdd8d22ea4aa99a4c84349056317cf550b6685e045e459954f258e590000001dbc890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
]
 Outs <- [
 0x0eb5ccb85c29009b6060dec353a38ea3b52cd2000000746a528800dbc890f77f49b96857648b72b77f9f82937f28a68704af05da0dc12ba53f2db
]
]

= [
 // typeID:
]
```

```

0x00, 0x00, 0x00, 0x00,
// networkID:
0x00, 0x00, 0x00, 0x04,
// blockchainID:
0x91, 0x06, 0x0e, 0xab, 0xfb, 0x5a, 0x57, 0x17,
0x20, 0x10, 0x9b, 0x58, 0x96, 0xe5, 0xff, 0x00,
0x01, 0xa0, 0x1c, 0xfe, 0x6b, 0x10, 0x3d, 0x58,
0x5e, 0x6e, 0xb9, 0x27, 0xb9, 0x7a, 0x17, 0x35,
// sourceChain:
0xd8, 0x91, 0xad, 0x56, 0x05, 0x6d, 0x9c, 0x01,
0xf1, 0x8f, 0x43, 0xf5, 0x8b, 0x5c, 0x78, 0x4a,
0xd0, 0x7a, 0x4a, 0x49, 0xcf, 0x3d, 0x1f, 0x11,
0x62, 0x38, 0x04, 0xb5, 0xcb, 0xa2, 0xc6, 0xbf,
// importedInputs[] count:
0x00, 0x00, 0x00, 0x01,
// importedInputs[0]
0x66, 0x13, 0xa4, 0x0d, 0xcd, 0xd8, 0xd2, 0x2e,
0xa4, 0xaa, 0x99, 0xa4, 0xc8, 0x43, 0x49, 0x05,
0x63, 0x17, 0xcf, 0x55, 0x0b, 0x66, 0x85, 0xe0,
0x45, 0xe4, 0x59, 0x95, 0x4f, 0x25, 0x8e, 0x59,
0x00, 0x00, 0x00, 0x01, 0xdb, 0xcf, 0x89, 0x0f,
0x77, 0xf4, 0x9b, 0x96, 0x85, 0x76, 0x48, 0xb7,
0x2b, 0x77, 0xf9, 0xf8, 0x29, 0x37, 0xf2, 0x8a,
0x68, 0x70, 0x4a, 0xf0, 0x5d, 0xa0, 0xdc, 0x12,
0xba, 0x53, 0xf2, 0xdb, 0x00, 0x00, 0x00, 0x05,
0x00, 0x00, 0x00, 0x74, 0x6a, 0x52, 0x88, 0x00,
0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
// outs[] count
0x00, 0x00, 0x00, 0x01,
// outs[0]
0x0e, 0xb5, 0xcc, 0xb8, 0x5c, 0x29, 0x00, 0x9b,
0x60, 0x60, 0xde, 0xcb, 0x35, 0x3a, 0x38, 0xea,
0x3b, 0x52, 0xcd, 0x20, 0x00, 0x00, 0x00, 0x74,
0x6a, 0x52, 0x88, 0x00, 0xdb, 0xcf, 0x89, 0x0f,
0x77, 0xf4, 0x9b, 0x96, 0x85, 0x76, 0x48, 0xb7,
0x2b, 0x77, 0xf9, 0xf8, 0x29, 0x37, 0xf2, 0x8a,
0x68, 0x70, 0x4a, 0xf0, 0x5d, 0xa0, 0xdc, 0x12,
0xba, 0x53, 0xf2, 0xdb,
]

```

## Credentials

Credentials have one possible type: `SECP256K1Credential`. Each credential is paired with an Input. The order of the credentials match the order of the inputs.

### SECP256K1 Credential

A `secp256k1` credential contains a list of 65-byte recoverable signatures.

#### What SECP256K1 Credential Contains

- `TypeID` is the ID for this type. It is `0x00000009`.
- `Signatures` is an array of 65-byte recoverable signatures. The order of the signatures must match the input's signature indices.

#### Gantt SECP256K1 Credential Specification

|                            |                                           |                                                 |
|----------------------------|-------------------------------------------|-------------------------------------------------|
| <code>+-----+-----+</code> | <code>  type_id : int  </code>            | <code>4 bytes  </code>                          |
| <code>+-----+-----+</code> | <code>  signatures : [] [65]byte  </code> | <code>4 + 65 * len(signatures) bytes  </code>   |
| <code>+-----+-----+</code> |                                           | <code>  8 + 65 * len(signatures) bytes  </code> |

#### Proto SECP256K1 Credential Specification

```

message SECP256K1Credential {
 uint32 typeID = 1; // 4 bytes
 repeated bytes signatures = 2; // 4 bytes + 65 bytes * len(signatures)
}

```

### SECP256K1 Credential Example

Let's make a payment input with:

- `TypeID` : 9
- `signatures` :

```
o 0x0acccf47a820549a84428440e2421975138790e41be262f7197f3d93faa26cc8741060d743ffaf025782c8c86b862d2b9febebe7d352f0b4591afbd1a
```

```
[
 TypeID <- 0x00000009
 Signatures <- [

0x0acccf47a820549a84428440e2421975138790e41be262f7197f3d93faa26cc8741060d743ffaf025782c8c86b862d2b9febebe7d352f0b4591afbd1a737f8a3(
]
]
=
[
 // Type ID
 0x00, 0x00, 0x00, 0x09,
 // length:
 0x00, 0x00, 0x00, 0x01,
 // sig[0]
 0xa0, 0xc0, 0xcf, 0x47, 0xa8, 0x20, 0x54, 0x9a,
 0x84, 0x42, 0x84, 0x40, 0xe2, 0x42, 0x19, 0x75,
 0x13, 0x87, 0x90, 0xe4, 0x1b, 0xe2, 0x62, 0xf7,
 0x19, 0x7f, 0x3d, 0x93, 0xfa, 0xa2, 0x6c, 0xc8,
 0x74, 0x10, 0x60, 0xd7, 0x43, 0xff, 0xaf, 0x02,
 0x57, 0x82, 0xc8, 0x86, 0x86, 0x2d, 0x2b,
 0x9f, 0xeb, 0xeb, 0xe7, 0xd3, 0x52, 0xf0, 0xb4,
 0x59, 0x1a, 0xfb, 0xd1, 0xa7, 0x37, 0xf8, 0xa3,
 0x00, 0x10, 0x19, 0x9d, 0xbf,
]
```

## Signed Transaction

A signed transaction contains an unsigned `AtomicTx` and credentials.

### What Signed Transaction Contains

A signed transaction contains a `CodecID`, `AtomicTx`, and `Credentials`.

- **CodecID** The only current valid codec id is `00 00`.
- **AtomicTx** is an atomic transaction, as described above.
- **Credentials** is an array of credentials. Each credential corresponds to the input at the same index in the `AtomicTx`

### Gantt Signed Transaction Specification

|                            |                                              |  |
|----------------------------|----------------------------------------------|--|
| +-----+-----+-----+        |                                              |  |
| codec_id : uint16          | 2 bytes                                      |  |
| +-----+-----+-----+        |                                              |  |
| atomic_tx : AtomicTx       | size(atomic_tx) bytes                        |  |
| +-----+-----+-----+        |                                              |  |
| credentials : []Credential | 4 + size(credentials) bytes                  |  |
| +-----+-----+-----+        |                                              |  |
|                            | 6 + size(atomic_tx) + len(credentials) bytes |  |
| +-----+-----+-----+        |                                              |  |

### Proto Signed Transaction Specification

```
message Tx {
 uint16 codec_id = 1; // 2 bytes
 AtomicTx atomic_tx = 2; // size(atomic_tx)
 repeated Credential credentials = 3; // 4 bytes + size(credentials)
}
```

### Signed Transaction Example

Let's make a signed transaction that uses the unsigned transaction and credential from the previous examples.

- **CodecID** : 0
  - **UnsignedTx** :  
0x0000000000000303991060eabfb5a571720109b5896e5ff00010a1cf6b103d585e6ebf27b97a1735d891ad56056d9c01f18f43f58b5c784ad07a4a49cf3d1;
  - **Credentials**
- ```
0x0000000900000010acccf47a820549a84428440e2421975138790e41be262f7197f3d93faa26cc8741060d743ffaf025782c8c86b862d2b9febebe7d352f
```

```
[  
    CodecID      <- 0x0000  
    UnsignedAtomic Tx <-
```

UTXO

A UTXO is a standalone representation of a transaction output.

What UTXO Contains

A UTXO contains a `CodecID`, `TxID`, `UTXOIndex`, `AssetID`, and `Output`.

- **CodecID** The only valid `CodecID` is `00 00`
 - **TxID** is a 32-byte transaction ID. Transaction IDs are calculated by taking sha256 of the bytes of the signed transaction.
 - **UTXOIndex** is an int that specifies which output in the transaction specified by `TxID` that this utxo was created by.
 - **AssetID** is a 32-byte array that defines which asset this utxo references.
 - **Output** is the output object that created this utxo. The serialization of Outputs was defined above.

Gantt UTXO Specification

```

+-----+-----+
| codec_id      : uint16    |          2 bytes |
+-----+-----+
| tx_id         : [32]byte  |          32 bytes |
+-----+-----+
| output_index  : int       |          4 bytes |
+-----+-----+

```

```

| asset_id      : [32]byte |           32 bytes |
+-----+-----+-----+
| output       : Output | size(output) bytes |
+-----+-----+-----+
| 70 + size(output) bytes |
+-----+

```

Proto UTXO Specification

```

message Utxo {
    uint16 codec_id = 1;      // 02 bytes
    bytes tx_id = 2;         // 32 bytes
    uint32 output_index = 3; // 04 bytes
    bytes asset_id = 4;      // 32 bytes
    Output output = 5;       // size(output)
}

```

UTXO Example

Let's make a UTXO from the signed transaction created above:

- **CodecID** : 0
- **TxID** : 0xf966750f438867c3c9828ddcbe660e21ccdbb36a9276958f011ba472f75d4e7
- **UTXOIndex** : 0 = 0x00000000
- **AssetID** : 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
- **Output** : "Example EVMOutput as defined above"

```

[
    CodecID   <- 0x0000
    TxID      <- 0xf966750f438867c3c9828ddcbe660e21ccdbb36a9276958f011ba472f75d4e7
    UTXOIndex <- 0x00000000
    AssetID   <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
    Output     <-
0x00000000700000000000000030390000000000000d4310000001000000251025c61fbfcf078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c894
]
=
[
    // Codec ID:
    0x00, 0x00,
    // txID:
    0xf9, 0x66, 0x75, 0x0f, 0x43, 0x88, 0x67, 0xc3,
    0xc9, 0x82, 0x8d, 0xdc, 0xdb, 0xe6, 0x60, 0xe2,
    0x1c, 0xcd, 0xbb, 0x36, 0xa9, 0x27, 0x69, 0x58,
    0xf0, 0x11, 0xba, 0x47, 0x2f, 0x75, 0xd4, 0xe7,
    // utxo index:
    0x00, 0x00, 0x00, 0x00,
    // assetID:
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0xb, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    // output:
    0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
    0x00, 0x00, 0x00, 0x02, 0x00, 0x01, 0x02, 0x03,
    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
    0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
    0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
    0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23,
    0x24, 0x25, 0x26, 0x27,
]

```

Cryptographic Primitives

Avalanche uses a variety of cryptographic primitives for its different functions. This file summarizes the type and kind of cryptography used at the network and blockchain layers.

Cryptography in the Network Layer

Avalanche uses Transport Layer Security, TLS, to protect node-to-node communications from eavesdroppers. TLS combines the practicality of public-key cryptography with the efficiency of symmetric-key cryptography. This has resulted in TLS becoming the standard for internet communication. Whereas most classical consensus protocols employ public-key cryptography to prove receipt of messages to third parties, the novel Snow* consensus family does not require

such proofs. This enables Avalanche to employ TLS in authenticating stakers and eliminates the need for costly public-key cryptography for signing network messages.

TLS Certificates

Avalanche does not rely on any centralized third-parties, and in particular, it does not use certificates issued by third-party authenticators. All certificates used within the network layer to identify endpoints are self-signed, thus creating a self-sovereign identity layer. No third parties are ever involved.

TLS Addresses

To avoid posting the full TLS certificate to the Platform chain, the certificate is first hashed. For consistency, Avalanche employs the same hashing mechanism for the TLS certificates as is used in Bitcoin. Namely, the DER representation of the certificate is hashed with sha256, and the result is then hashed with ripemd160 to yield a 20-byte identifier for stakers.

This 20-byte identifier is represented by "NodeID-" followed by the data's [CB58](#) encoded string.

Cryptography in the Avalanche Virtual Machine

The Avalanche virtual machine uses elliptic curve cryptography, specifically `secp256k1`, for its signatures on the blockchain.

This 32-byte identifier is represented by "PrivateKey-" followed by the data's [CB58](#) encoded string.

Secp256k1 Addresses

Avalanche is not prescriptive about addressing schemes, choosing to instead leave addressing up to each blockchain.

The addressing scheme of the X-Chain and the P-Chain relies on `secp256k1`. Avalanche follows a similar approach as Bitcoin and hashes the ECDSA public key. The 33-byte compressed representation of the public key is hashed with sha256 **once**. The result is then hashed with ripemd160 to yield a 20-byte address.

Avalanche uses the convention `chainID-address` to specify which chain an address exists on. `chainID` may be replaced with an alias of the chain. When transmitting information through external applications, the CB58 convention is required.

Bech32

Addresses on the X-Chain and P-Chain use the [Bech32](#) standard outlined in [BIP 0173](#). There are four parts to a Bech32 address scheme. In order of appearance:

- A human-readable part (HRP). On Mainnet this is `avax`.
- The number `1`, which separates the HRP from the address and error correction code.
- A base-32 encoded string representing the 20 byte address.
- A 6-character base-32 encoded error correction code.

Additionally, an Avalanche address is prefixed with the alias of the chain it exists on, followed by a dash. For example, X-Chain addresses are prefixed with `x-`.

The following regular expression matches addresses on the X-Chain, P-Chain and C-Chain for Mainnet, Fuji and localhost. Note that all valid Avalanche addresses will match this regular expression, but some strings that are not valid Avalanche addresses may match this regular expression.

```
^( [XPC] | [a-km-zA-HJ-NP-Z1-9] {36,72} ) - [a-zA-Z] {1,83} 1[qpzry9x8gf2tvdw0s3jn54khce6mua71] {38} $
```

Read more about Avalanche's [addressing scheme](#).

Secp256k1 Recoverable Signatures

Recoverable signatures are stored as the 65-byte `[r || s || v]` where `v` is 0 or 1 to allow quick public key recoverability. `s` must be in the lower half of the possible range to prevent signature malleability. Before signing a message, the message is hashed using sha256.

Secp256k1 Example

Suppose Rick and Morty are setting up a secure communication channel. Morty creates a new public-private key pair.

Private Key: `0x98cb077f972feb0481f1d894f272c6ale3c15e272a1658ff716444f46520070`

Public Key (33-byte compressed): `0x02b33c917f2f6103448d7feb42614037d05928433cb25e78f01a825aa829bb3c27`

Because of Rick's infinite wisdom, he doesn't trust himself with carrying around Morty's public key, so he only asks for Morty's address. Morty follows the instructions, SHA256's his public key, and then ripemd160's that result to produce an address.

SHA256(Public Key): `0x28d7670d71667e93ff586f664937f52828e6290068fa2a37782045bffa7b0d2f`

Address: `0xe8777f38c88ca153a6fdc25942176d2bf5491b89`

Morty is quite confused because a public key should be safe to be public knowledge. Rick belches and explains that hashing the public key protects the private key owner from potential future security flaws in elliptic curve cryptography. In the event cryptography is broken and a private key can be derived from a public key, users can transfer their funds to an address that has never signed a transaction before, preventing their funds from being compromised by an attacker. This enables coin owners to be protected while the cryptography is upgraded across the clients.

Later, once Morty has learned more about Rick's backstory, Morty attempts to send Rick a message. Morty knows that Rick will only read the message if he can verify it was from him, so he signs the message with his private key.

Message: `0x68656c702049276d207472617070656420696e206120636f6d7075746572`

Message Hash: `0x912800c29d554fb9cdce579c0abba991165bbbc8bfec9622481d01e0b3e4b7da`

Message Signature:

```
0xb52aa0535c5c48268d843bd65395623d2462016325a86f09420c81f142578e121d11bd368b88ca6de4179a007e6abe0e8d0be1a6a4485def8f9e02957d3d72da01
```

Morty was never seen again.

Signed Messages

A standard for interoperable generic signed messages based on the Bitcoin Script format and Ethereum format.

```
sign(sha256(length(prefix) + prefix + length(message) + message))
```

The prefix is simply the string `\x1AAvalanche Signed Message:\n`, where `0x1A` is the length of the prefix text and `length(message)` is an [integer](#) of the message size.

Gantt Pre-Image Specification

prefix : [26]byte	26 bytes
messageLength : int	4 bytes
message : []byte	size(message) bytes
	26 + 4 + size(message)

Example

As an example we will sign the message "Through consensus to the stars"

```
// prefix size: 26 bytes
0x1a
// prefix: Avalanche Signed Message:\n
0x41 0x76 0x61 0x6c 0x61 0x6e 0x63 0x68 0x65 0x20 0x53 0x69 0x67 0x6e 0x65 0x64 0x20 0x4d 0x65 0x73 0x73 0x61 0x67 0x65 0x3a
0x0a
// msg size: 30 bytes
0x00 0x00 0x00 0x1e
// msg: Through consensus to the stars
54 68 72 6f 75 67 68 20 63 6f 6e 73 65 6e 73 75 73 20 74 6f 20 74 68 65 20 73 74 61 72 73
```

After hashing with `sha256` and signing the pre-image we return the value [cb58](#) encoded:

`4Eb2zAHF4JjZPFJmp4usSokTGqq9mEGwVMY2WZzzCmu657SNFZhndsiS8TvL32n3bexd8emUwiXs8XqKjhqzvoRFvghnvSN`. Here's an example using the [Avalanche Web Wallet](#).

 Sign message

Cryptography in Ethereum Virtual Machine

Avalanche nodes support the full Ethereum Virtual Machine (EVM) and precisely duplicate all of the cryptographic constructs used in Ethereum. This includes the Keccak hash function and the other mechanisms used for cryptographic security in the EVM.

Cryptography in Other Virtual Machines

Since Avalanche is an extensible platform, we expect that people will add additional cryptographic primitives to the system over time.

Network Protocol

Avalanche network defines the core communication format between Avalanche nodes. It uses the [primitive serialization](#) format for payload packing.

"Containers" are mentioned extensively in the description. A Container is simply a generic term for blocks or vertices, without needing to specify whether the consensus algorithm is DAG or Chain.

GetVersion

`GetVersion` requests for a `Version` message to be sent as a response.

The OpCode used by `GetVersion` messages is: `0x00`.

What GetVersion Contains

The payload of a `GetVersion` message is empty.

[]

How GetVersion Is Handled

A node receiving a `GetVersion` message must respond with a `Version` message containing the current time and node version.

When GetVersion Is Sent

`GetVersion` is sent when a node is connected to another node, but has not yet received a `Version` message. It may, however, be re-sent at any time.

Version

`Version` ensures that the nodes we are connected to are running compatible versions of Avalanche, and at least loosely agree on the current time.

The OpCode used by `Version` messages is: `0x01`.

What Version Contains

`Version` contains the node's current time in Unix time format in number of milliseconds since the beginning of the epoch in January 1, 1970, as well as a version string describing the version of the code that the node is running.

Content:

```
[  
    Long    <- Unix Timestamp (Seconds)  
    String <- Version String  
]
```

How Version Is Handled

If the versions are incompatible or the current times differ too much, the connection will be terminated.

When Version Is Sent

`Version` is sent in response to a `GetVersion` message.

Version Example

Sending a `Version` message with the time November 16th, 2008 at 12:00am (UTC) and the version `avalanche/0.0.1`

```
[  
    Long    <- 1226793600 = 0x00000000491f6280  
    String <- "avalanche/0.0.1"  
]  
=  
[  
    0x00, 0x00, 0x00, 0x00, 0x49, 0x1f, 0x62, 0x80,  
    0x00, 0x0f, 0x61, 0x76, 0x61, 0x6c, 0x61, 0x6e,  
    0x63, 0x68, 0x65, 0x2f, 0x30, 0x2e, 0x30, 0x2e,  
    0x31,  
]
```

GetPeers

Overview

`GetPeers` requests that a `Peers` message be sent as a response.

The OpCode used by `GetPeers` messages is: `0x02`.

What GetPeers Contains

The payload of a `GetPeers` message is empty.

```
[]
```

How GetPeers Is Handled

A node receiving `GetPeers` request must respond with a `Peers` message containing the IP addresses of its connected, staking nodes.

When GetPeers Is Sent

A node sends `GetPeers` messages upon startup to discover the participants in the network. It may also periodically send `GetPeers` messages in order to discover new nodes as they arrive in the network.

Peers

Overview

`Peers` message contains a list of peers, represented as IP Addresses. Note that an IP Address contains both the IP and the port number, and supports both IPv4 and IPv6 format.

The OpCode used by `Peers` messages is: `0x03`.

What Peers Contains

`Peers` contains the IP addresses of the staking nodes this node is currently connected to.

Content:

```
[  
    Variable Length IP Address Array  
]
```

How Peers Is Handled

On receiving a `Peers` message, a node should compare the nodes appearing in the message to its own list of neighbors, and forge connections to any new nodes.

When Peers Is Sent

`Peers` messages do not need to be sent in response to a `GetPeers` message, and are sent periodically to announce newly arriving nodes. The default period for such push gossip is 60 seconds.

Peers Example

Sending a `Peers` message with the IP addresses "127.0.0.1:9650" and "[2001:0db8:ac10:fe01::]:12345"

```
[  
    Variable Length IP Address Array <- ["127.0.0.1:9650", "[2001:0db8:ac10:fe01::]:12345"]  
]  
=  
[  
    0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,  
    0x7f, 0x00, 0x00, 0x01, 0x25, 0xb2, 0x20, 0x01,  
    0xd, 0xb8, 0xac, 0x10, 0xfe, 0x01, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x39,  
]
```

Get

Overview

A `Get` message requests a container, that is, block or vertex, from a node.

The OpCode used by `Get` messages is: `0x04`.

What Get Contains

A `Get` message contains a `SubnetID`, `RequestID`, and `ContainerID`.

`SubnetID` defines which Subnets this message is destined for.

`RequestID` is a counter that helps keep track of the messages sent by a node. Each time a node sends an un-prompted message, the node will create a new unique `RequestID` for the message.

`ContainerID` is the identifier of the requested container.

```
[  
    Length 32 Byte Array <- SubnetID  
    UInt             <- RequestID  
    Length 32 Byte Array <- ContainerID  
]
```

How Get Is Handled

The node should reply with a `Put` message with the same `SubnetID`, `RequestID`, and `ContainerID` along with the `Container` with the specified identifier. Under correct situations, a node should only be asked for a container that it has. Therefore, if the node does not have the specified container, the `Get` message can safely be dropped.

When Get Is Sent

A node will send a `Get` message to a node that tells us about the existence of a container. For example, suppose we have two nodes: Rick and Morty. If Rick sends a `PullQuery` message that contains a `ContainerID`, that Morty doesn't have the container for, then Morty will send a `Get` message containing the missing `ContainerID`.

Get Example

```
[  
    SubnetID     <- 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
```

```

RequestID    <- 43110 = 0x0000A866
ContainerID <- 0x2122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f40
]
=
[
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
  0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10,
  0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
  0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20,
  0x00, 0x00, 0xa8, 0x66, 0x21, 0x22, 0x23, 0x24,
  0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c,
  0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34,
  0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c,
  0x3d, 0x3e, 0x3f, 0x40,
]

```

Put

Overview

A `Put` message provides a requested container to a node.

The OpCode used by `Put` messages is: `0x05`.

What Put Contains

A `Put` message contains a `SubnetID`, `RequestID`, `ContainerID`, and `Container`.

`SubnetID` defines which Subnets this message is destined for.

`RequestID` is a counter that helps keep track of the messages sent by a node.

`ContainerID` is the identifier of the container this message is sending.

`Container` is the bytes of the container this message is sending.

```

[
  Length 32 Byte Array      <- SubnetID
  UInt                  <- RequestID
  Length 32 Byte Array      <- ContainerID
  Variable Length Byte Array <- Container
]
```

How Put Is Handled

The node should attempt to add the container to consensus.

When Put Is Sent

A node will send a `Put` message in response to receiving a Get message for a container the node has access to.

Put Example

```

[
  SubnetID    <- 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
  RequestID   <- 43110 = 0x0000A866
  ContainerID <- 0x5ba080dcf6861c94c24ec62bc09a3c8b0fdd4691ebf02491e0e921dd0c77206f
  Container   <- 0x2122232425
]
=
[
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
  0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10,
  0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
  0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20,
  0x00, 0x00, 0xa8, 0x66, 0x21, 0x22, 0x23, 0x24,
  0x25, 0x26, 0x27, 0x28, 0x29, 0x2a, 0x2b, 0x2c,
  0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34,
  0x35, 0x36, 0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c,
  0x3d, 0x3e, 0x3f, 0x40,
]
```

PushQuery

Overview

A `PushQuery` message requests the preferred `containerIDs` from the node after the specified `ContainerID` has been added to consensus. If the `ContainerID` is not known, the `Container` is optimistically provided.

The OpCode used by PushQuery messages is: 0x06 .

What PushQuery Contains

A PushQuery message contains a SubnetID, RequestID, ContainerID, and Container.

SubnetID defines which Subnets this message is destined for.

RequestID is a counter that helps keep track of the messages sent by a node.

ContainerID is the identifier of the container this message expects to have been added to consensus before the response is sent.

Container is the bytes of the container with identifier **ContainerID**.

```
[  
    Length 32 Byte Array      <- SubnetID  
    UInt                      <- RequestID  
    Length 32 Byte Array      <- ContainerID  
    Variable Length Byte Array <- Container  
]
```

How PushQuery Is Handled

The node should attempt to add the container to consensus. After the container is added to consensus, a `Chits` message should be sent with the current preferences of the node.

When PushQuery Is Sent

A node should send a `PushQuery` message if it wants to learn of this node's current preferences and it feels that it is possible the node hasn't learned of `Container` yet. The node will want to learn of nodes preferences when it learns of a new container or it has had pending containers for "awhile."

PushQuery Example

```
SubnetID    <- 0x010203040506070809a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
RequestID   <- 43110 = 0x0000A866
ContainerID <- 0x5ba080dcf6861c94c24ec62bc09a3c8b0fdd4691ebf02491e0e921dd0c77206f
Container    <- 0x2122232425

]
=
[

0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf, 0x10,
0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20,
0x00, 0x00, 0xa8, 0x66, 0x5b, 0xa0, 0x80, 0xdc,
0xf6, 0x86, 0x1c, 0x94, 0xc2, 0x4e, 0xc6, 0x2b,
0xc0, 0x9a, 0x3c, 0x8b, 0x0f, 0xdd, 0x46, 0x91,
0xeb, 0xf0, 0x24, 0x91, 0xe0, 0xe9, 0x21, 0xdd,
0x0c, 0x77, 0x20, 0x6f, 0x00, 0x00, 0x00, 0x05,
0x21, 0x22, 0x23, 0x24, 0x25,
]

]
```

PullQuery

Overview

A `PullQuery` message requests the preferred container IDs from the node after the specified `ContainerID` has been added to consensus.

The OpCode used by PullQuery messages is: 0x07 .

What PullQuery Contains

A PullQuery message contains a SubnetID, RequestID, and ContainerID.

SubnetID defines which Subnets this message is destined for.

RequestID is a counter that helps keep track of the messages sent by a node.

Container-ID is the identifier of the container this message expects to have been added to `conscious` before the response is sent.

```
[  
    Length 32 Byte Array <- SubnetID
```

```

Length 32 Byte Array <- ContainerID
]

```

How PullQuery Is Handled

If the node hasn't added `ContainerID`, it should attempt to add the container to consensus. After the container is added to consensus, a `Chits` message should be sent with the current preferences of the node.

When PullQuery Is Sent

A node should send a `PullQuery` message if it wants to learn of this node's current preferences and it feels that it quite likely the node has already learned of `Container`. The node will want to learn of nodes preferences when it learns of a new container or it has had pending containers for "awhile."

PullQuery Example

```

[
    SubnetID    <- 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
    RequestID   <- 43110 = 0x0000A866
    ContainerID <- 0x5ba080dcf6861c94c24ec62bc09a3c8b0fdd4691ebf02491e0e921dd0c77206f
]
=
[
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10,
    0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
    0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20,
    0x00, 0x01, 0xa8, 0x66, 0x5b, 0xa0, 0x80, 0xdc,
    0xf6, 0x86, 0x1c, 0x94, 0xc2, 0x4e, 0xc6, 0x2b,
    0xc0, 0x9a, 0x3c, 0xb8, 0x0f, 0xdd, 0x46, 0x91,
    0xeb, 0xf0, 0x24, 0x91, 0xe0, 0xe9, 0x21, 0xdd,
    0x0c, 0x77, 0x20, 0x6f,
]

```

Chits

Overview

A `Chits` message provides a requested set of preferred containers to a node.

The OpCode used by `Chits` messages is: `0x08`.

What Chits Contains

A `Chits` message contains a `SubnetID`, `RequestID`, and `Preferences`.

`SubnetID` defines which Subnets this message is destined for.

`RequestID` is a counter that helps keep track of the messages sent by a node.

`Preferences` is the list of `containerIDs` that fully describe the node's preferences.

```

[
    Length 32 Byte Array           <- SubnetID
    UInt                          <- RequestID
    Variable Length (Length 32 Byte Array) Array <- Preferences
]

```

How Chits Is Handled

The node should attempt to add any referenced containers to consensus. If the referenced containers can't be added, the node can ignore the missing containers and apply the remaining chits to the poll. Once a poll is completed, container confidences should be updated appropriately.

When Chits Is Sent

A node will send a `Chits` message in response to receiving a `PullQuery` or `PushQuery` message for a container the node has added to consensus.

Chits Example

```

[
    SubnetID    <- 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20
    RequestID   <- 43110 = 0x0000A866
    Preferences <- [
        0x2122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f40,
        0x4142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f60,
    ]
]
=
```

```
[  
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,  
    0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10,  
    0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,  
    0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20,  
    0x00, 0x00, 0xa8, 0x66, 0x00, 0x00, 0x00, 0x02,  
    0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28,  
    0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f, 0x30,  
    0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38,  
    0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40,  
    0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48,  
    0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f, 0x50,  
    0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,  
    0x59, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60,  
]  
]
```

Platform Transaction Format

This file is meant to be the single source of truth for how we serialize transactions in Avalanche's Platform Virtual Machine, aka the `Platform Chain` or `P-Chain`. This document uses the [primitive serialization](#) format for packing and [secp256k1](#) for cryptographic user identification.

Codec ID

Some data is prepended with a codec ID (uint16) that denotes how the data should be deserialized. Right now, the only valid codec ID is 0 (`0x00 0x00`).

Proof of Possession

A BLS public key and a proof of possession of the key.

What Proof of Possession Contains

- **PublicKey** is the 48 byte representation of the public key.
- **Signature** is the 96 byte signature by the private key over its public key.

Proof of Possession Specification

+-----+	+-----+	+-----+
public_key : [48]byte	48 bytes	
+-----+	+-----+	+-----+
signature : [96]byte	96 bytes	
+-----+	+-----+	+-----+
	144 bytes	
+-----+	+-----+	+-----+

Proof of Possession Specification

```
message ProofOfPossession {
    bytes public_key = 1; // 48 bytes
    bytes signature = 2; // 96 bytes
}
```

Proof of Possession Example

```
// Public Key:  
0x85, 0x02, 0x5b, 0xca, 0x6a, 0x30, 0x2d, 0xc6,  
0x13, 0x38, 0xff, 0x49, 0xc8, 0xba, 0xa5, 0x72,  
0xde, 0xd3, 0xe8, 0xf, 0x37, 0x59, 0x30, 0x4c,  
0x7f, 0x61, 0x8a, 0x2a, 0x25, 0x93, 0xc1, 0x87,  
0x0, 0x80, 0xa3, 0xcf, 0xde, 0xc9, 0x50, 0x40,  
0x30, 0x9a, 0xd1, 0xf1, 0x58, 0x95, 0x30, 0x67,  
// Signature:  
0xb, 0x1d, 0x61, 0x33, 0xd1, 0x7e, 0x34, 0x83,  
0x22, 0x0a, 0xd9, 0x60, 0xb6, 0xfd, 0x1, 0x1e,  
0x4e, 0x12, 0x14, 0xa8, 0xce, 0x21, 0xef, 0x61,  
0x62, 0x27, 0xe5, 0xd5, 0xee, 0xf0, 0x70, 0xd7,  
0x50, 0x0e, 0x6f, 0x7d, 0x44, 0x52, 0xc5, 0xa7,  
0x60, 0x62, 0x0c, 0xc0, 0x67, 0x95, 0xcb, 0xe2,  
0x18, 0xe0, 0x72, 0xeb, 0xa7, 0x6d, 0x94, 0x78,  
0x8d, 0x9d, 0x01, 0x17, 0x6c, 0xe4, 0xec, 0xad,  
0xfb, 0x96, 0xb4, 0x7f, 0x94, 0x22, 0x81, 0x89,  
0x4d, 0xdf, 0xad, 0xd1, 0xc1, 0x74, 0x3f, 0x7f,  
0x54, 0x9f, 0x1d, 0x07, 0xd5, 0x9d, 0x55, 0x65,  
0x59, 0x27, 0xf7, 0x2b, 0xc6, 0xb, 0x7c, 0x12
```

Transferable Output

Transferable outputs wrap an output with an asset ID.

What Transferable Output Contains

A transferable output contains an `AssetID` and an `Output`.

- **AssetID** is a 32-byte array that defines which asset this output references. The only valid `AssetID` is the AVAX `AssetID`.
 - **Output** is an output, as defined below. For example, this can be a SECP256K1 transfer output.

Gantt Transferable Output Specification

```
+-----+-----+
| asset_id : [32]byte |           32 bytes |
+-----+
| output   : Output    |     size(output) bytes |
+-----+
| 32 + size(output) bytes |
+-----+
```

Proto Transferable Output Specification

```
message TransferableOutput {
    bytes asset_id = 1; // 32 bytes
    Output output = 2; // size(output)
}
```

Transferable Output Example

Let's make a transferable output:

- AssetID: 0x6870b7d66ac32540311379e5b5bdbad28ec7eb8ddfbfc8f4d67299ebba48475907a
 - Output: "Example SECP256K1 Transfer Output from below"

Transferable Input

Transferable inputs describe a specific UTXO with a provided transfer input

What Transferable Input Contains

A transferable input contains a TxID, UTXOIndex, AssetID and an Input.

- `TxID` is a 32-byte array that defines which transaction this input is consuming an output from.
 - `UTXOIndex` is an int that defines which utxo this input is consuming the specified transaction.
 - `AssetID` is a 32-byte array that defines which asset this input references. The only valid `AssetID` is the AVAX `AssetID`.
 - `Input` is a transferable input object.

Gantt Transferable Input Specification

```
+-----+-----+
| tx_id      : [32]byte |           32 bytes |
+-----+-----+
| utxo_index : int        |          04 bytes |
+-----+-----+
```

```

| asset_id : [32]byte |           32 bytes |
+-----+-----+
| input   : Input    | size(input) bytes |
+-----+-----+
| 68 + size(input) bytes |
+-----+

```

Proto Transferable Input Specification

```

message TransferableInput {
    bytes tx_id = 1;           // 32 bytes
    uint32 utxo_index = 2;     // 04 bytes
    bytes asset_id = 3;        // 32 bytes
    Input input = 4;           // size(input)
}

```

Transferable Input Example

Let's make a transferable input:

- **TxID** : 0x0dfafbd5c81f635c9257824ff21c8e3e6f7b632ac306e11446ee540d34711a15
- **UTXOIndex** : 0
- **AssetID** : 0x6870b7d66ac32540311379e5b5dbad28ec7eb8ddbfc8f4d67299ebb48475907a
- **Input** : "Example SECP256K1 Transfer Input from below"

```

[
    TxID      <- 0x0dfafbd5c81f635c9257824ff21c8e3e6f7b632ac306e11446ee540d34711a15
    UTXOIndex <- 0x00000001
    AssetID   <- 0x6870b7d66ac32540311379e5b5dbad28ec7eb8ddbfc8f4d67299ebb48475907a
    Input      <- 0x0000000500000000e6b28000000000100000000
]
=
[
    // txID:
    0xdf, 0xaf, 0xbd, 0xf5, 0xc8, 0x1f, 0x63, 0x5c,
    0x92, 0x57, 0x82, 0x4f, 0x2, 0x1c, 0x8e, 0x3e,
    0x6f, 0x7b, 0x63, 0x2a, 0xc3, 0x06, 0x1, 0x14,
    0x46, 0xee, 0x54, 0xd, 0x34, 0x71, 0x1a, 0x15,
    // utxoIndex:
    0x00, 0x00, 0x00, 0x01,
    // assetID:
    0x68, 0x70, 0xb7, 0xd6, 0x6a, 0xc3, 0x25, 0x40,
    0x31, 0x13, 0x79, 0xe5, 0xb5, 0xdb, 0xad, 0x28,
    0xec, 0x7e, 0xb8, 0xdd, 0xbf, 0xc8, 0xf4, 0xd6,
    0x72, 0x99, 0xeb, 0xb4, 0x84, 0x75, 0x90, 0x7a,
    // input:
    0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
    0xee, 0x6b, 0x28, 0x00, 0x00, 0x00, 0x00, 0x01,
    0x00, 0x00, 0x00, 0x00
]

```

Outputs

Outputs have two possible type: `SECP256K1TransferOutput` , `SECP256K1OutputOwners` .

SECP256K1 Transfer Output

A [secp256k1](#) transfer output allows for sending a quantity of an asset to a collection of addresses after a specified Unix time. The only valid asset is AVAX.

What SECP256K1 Transfer Output Contains

A secp256k1 transfer output contains a `TypeID` , `Amount` , `Locktime` , `Threshold` , and `Addresses` .

- **TypeID** is the ID for this output type. It is `0x00000007` .
- **Amount** is a long that specifies the quantity of the asset that this output owns. Must be positive.
- **Locktime** is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- **Threshold** is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses` . If `Addresses` is empty, must be 0.
- **Addresses** is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

Gantt SECP256K1 Transfer Output Specification

```

+-----+-----+-----+
| type_id : int |           4 bytes |
+-----+

```

```

+-----+-----+
| amount : long | 8 bytes |
+-----+-----+
| locktime : long | 8 bytes |
+-----+-----+
| threshold : int | 4 bytes |
+-----+-----+
| addresses : [] [20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+
| 28 + 20 * len(addresses) bytes |
+-----+-----+

```

Proto SECP256K1 Transfer Output Specification

```

message SECP256K1TransferOutput {
    uint32 type_id = 1; // 04 bytes
    uint64 amount = 2; // 08 bytes
    uint64 locktime = 3; // 08 bytes
    uint32 threshold = 4; // 04 bytes
    repeated bytes addresses = 5; // 04 bytes + 20 bytes * len(addresses)
}

```

SECP256K1 Transfer Output Example

Let's make a secp256k1 transfer output with:

- **TypeID** : 7
- **Amount** : 3999000000
- **Locktime** : 0
- **Threshold** : 1
- **Addresses** :
 - 0xda2bee01be82ecc00c34f361eda8eb30fb5a715c

```

[
    TypeID    <- 0x00000007
    Amount    <- 0x00000000ee5be5c0
    Locktime <- 0x0000000000000000
    Threshold <- 0x00000001
    Addresses <- [
        0xda2bee01be82ecc00c34f361eda8eb30fb5a715c,
    ]
]
=
[
    // type_id:
    0x00, 0x00, 0x00, 0x07,
    // amount:
    0x00, 0x00, 0x00, 0x00, 0xee, 0x5b, 0xe5, 0xc0,
    // locktime:
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    // threshold:
    0x00, 0x00, 0x00, 0x01,
    // number of addresses:
    0x00, 0x00, 0x00, 0x01,
    // addrs[0]:
    0xda, 0x2b, 0xee, 0x01, 0xbe, 0x82, 0xec, 0xc0,
    0x0c, 0x34, 0xf3, 0x61, 0xed, 0xa8, 0xeb, 0x30,
    0xfb, 0x5a, 0x71, 0x5c,
]

```

SECP256K1 Output Owners Output

A [secp256k1](#) output owners output will receive the staking rewards when the lock up period ends.

What SECP256K1 Output Owners Output Contains

A secp256k1 output owners output contains a `TypeID`, `Locktime`, `Threshold`, and `Addresses`.

- **TypeID** is the ID for this output type. It is `0x0000000b`.
- **Locktime** is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- **Threshold** is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`.
- If `Addresses` is empty, must be 0.
- `Addresses` is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

Gantt SECP256K1 Output Owners Output Specification

```
+-----+-----+-----+
| type_id : int |          4 bytes |
+-----+-----+-----+
| locktime : long |          8 bytes |
+-----+-----+-----+
| threshold : int |          4 bytes |
+-----+-----+-----+
| addresses : [][20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+-----+
| 20 + 20 * len(addresses) bytes |
+-----+
```

Proto SECP256K1 Output Owners Output Specification

```
message SECP256K1OutputOwnersOutput {
    uint32 type_id = 1;           // 04 bytes
    uint64 locktime = 2;          // 08 bytes
    uint32 threshold = 3;         // 04 bytes
    repeated bytes addresses = 4; // 04 bytes + 20 bytes * len(addresses)
}
```

SECP256K1 Output Owners Output Example

Let's make a secp256k1 output owners output with:

- **TypeID** : 11
- **Locktime** : 0
- **Threshold** : 1
- **Addresses** :
 - 0xda2bee01be82ecc00c34f361eda8eb30fb5a715c

```
[  
    TypeID    <- 0x0000000b  
    Locktime  <- 0x0000000000000000  
    Threshold <- 0x00000001  
    Addresses <- [  
        0xda2bee01be82ecc00c34f361eda8eb30fb5a715c,  
    ]  
]  
=  
[  
    // type_id:  
    0x00, 0x00, 0x00, 0x0b,  
    // locktime:  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    // threshold:  
    0x00, 0x00, 0x00, 0x01,  
    // number of addresses:  
    0x00, 0x00, 0x00, 0x01,  
    // addrs[0]:  
    0xda, 0x2b, 0xee, 0x01, 0xbe, 0x82, 0xec, 0xc0,  
    0x0c, 0x34, 0xf3, 0x61, 0xed, 0xa8, 0xeb, 0x30,  
    0xfb, 0x5a, 0x71, 0x5c,  
]
```

Inputs

Inputs have one possible type: `SECP256K1TransferInput`.

SECP256K1 Transfer Input

A `secp256k1` transfer input allows for spending an unspent secp256k1 transfer output.

What SECP256K1 Transfer Input Contains

A secp256k1 transfer input contains an `Amount` and `AddressIndices`.

- **TypeID** is the ID for this output type. It is `0x00000005`.
- **Amount** is a long that specifies the quantity that this input should be consuming from the UTXO. Must be positive. Must be equal to the amount specified in the UTXO.
- **AddressIndices** is a list of unique ints that define the private keys are being used to spend the UTXO. Each UTXO has an array of addresses that can spend the UTXO. Each int represents the index in this address array that will sign this transaction. The array must be sorted low to high.

Gantt SECP256K1 Transfer Input Specification

```
+-----+-----+
| type_id      : int |          4 bytes |
+-----+-----+
| amount       : long |          8 bytes |
+-----+-----+
| address_indices : []int |  4 + 4 * len(address_indices) bytes |
+-----+-----+
|               | 16 + 4 * len(address_indices) bytes |
+-----+-----+
```

Proto SECP256K1 Transfer Input Specification

```
message SECP256K1TransferInput {
    uint32 type_id = 1;           // 04 bytes
    uint64 amount = 2;           // 08 bytes
    repeated uint32 address_indices = 3; // 04 bytes + 4 bytes * len(address_indices)
}
```

SECP256K1 Transfer Input Example

Let's make a payment input with:

- **TypeID** : 5
- **Amount** : 4000000000
- **AddressIndices** : [0]

```
[
    TypeID      <- 0x00000005
    Amount       <- 0x00000000ee6b2800
    AddressIndices <- [0x00000000]
]
=
[
    // type_id:
    0x00, 0x00, 0x00, 0x05,
    // amount:
    0x00, 0x00, 0x00, 0x00, 0xee, 0x6b, 0x28, 0x00,
    // length:
    0x00, 0x00, 0x00, 0x01,
    // address_indices[0]
    0x00, 0x00, 0x00, 0x00
]
```

Unsigned Transactions

Unsigned transactions contain the full content of a transaction with only the signatures missing. Unsigned transactions have six possible types:

`AddValidatorTx`, `AddSubnetValidatorTx`, `AddDelegatorTx`, `CreateSubnetTx`, `ImportTx`, and `ExportTx`. They embed `BaseTx`, which contains common fields and operations.

Unsigned BaseTx

What Base TX Contains

A base TX contains a `TypeID`, `NetworkID`, `BlockchainID`, `Outputs`, `Inputs`, and `Memo`.

- **TypeID** is the ID for this type. It is `0x00000000`.
- **NetworkID** is an int that defines which network this transaction is meant to be issued to. This value is meant to support transaction routing and is not designed for replay attack prevention.
- **BlockchainID** is a 32-byte array that defines which blockchain this transaction was issued to. This is used for replay attack prevention for transactions that could potentially be valid across network or blockchain.
- **Outputs** is an array of transferable output objects. Outputs must be sorted lexicographically by their serialized representation. The total quantity of the assets created in these outputs must be less than or equal to the total quantity of each asset consumed in the inputs minus the transaction fee.
- **Inputs** is an array of transferable input objects. Inputs must be sorted and unique. Inputs are sorted first lexicographically by their `TxID` and then by the `UTXOIndex` from low to high. If there are inputs that have the same `TxID` and `UTXOIndex`, then the transaction is invalid as this would result in a double spend.
- **Memo** Memo field contains arbitrary bytes, up to 256 bytes.

Gantt Base TX Specification

```
+-----+-----+
| type_id      : int |          4 bytes |
+-----+-----+
```

```

| network_id      : int           |          4 bytes |
+-----+
| blockchain_id : [32]byte       |          32 bytes |
+-----+
| outputs        : []TransferableOutput | 4 + size(outputs) bytes |
+-----+
| inputs         : []TransferableInput  | 4 + size(inputs) bytes |
+-----+
| memo           : [256]byte       | 4 + size(memo) bytes |
+-----+
|                                     | 52 + size(outputs) + size(inputs) + size(memo) bytes |
+-----+

```

Proto Base TX Specification

```
message BaseTx {
    uint32 type_id = 1;           // 04 bytes
    uint32 network_id = 2;        // 04 bytes
    bytes blockchain_id = 3;      // 32 bytes
    repeated Output outputs = 4;  // 04 bytes + size(outs)
    repeated Input inputs = 5;    // 04 bytes + size(ins)
    bytes memo = 6;              // 04 bytes + size(memo)
}
```

Base TX Example

Let's make a base TX that uses the inputs and outputs from the previous examples:

```
// transferable input:  
0xdf, 0xaf, 0xbd, 0xf5, 0xc8, 0x1f, 0x63, 0x5c,  
0x92, 0x57, 0x82, 0x4f, 0xf2, 0x1c, 0x8e, 0x3e,  
0x6f, 0x7b, 0x63, 0x2a, 0xc3, 0x06, 0xe1, 0x14,  
0x46, 0xee, 0x54, 0x0d, 0x34, 0x71, 0x1a, 0x15,  
0x00, 0x00, 0x00, 0x01,  
0x68, 0x70, 0xb7, 0xd6, 0x6a, 0xc3, 0x25, 0x40,  
0x31, 0x13, 0x79, 0xe5, 0xb5, 0xdb, 0xad, 0x28,  
0xec, 0x7e, 0xb8, 0xd0, 0xbf, 0xc8, 0xf4, 0xd6,  
0x72, 0x99, 0xeb, 0xb4, 0x84, 0x75, 0x90, 0x7a,  
0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,  
0xee, 0x6b, 0x28, 0x00, 0x00, 0x00, 0x00, 0x01,  
0x00, 0x00, 0x00, 0x00,  
// Memo length:  
0x00, 0x00, 0x00, 0x00,  
]
```

Unsigned Add Validator TX

What Unsigned Add Validator TX Contains

An unsigned add validator TX contains a BaseTx , Validator , Stake , RewardsOwner , and Shares . The TypeID for this type is 0x0000000c .

- **BaseTx**
 - **Validator** Validator has a `NodeID` , `StartTime` , `EndTime` , and `Weight`
 - `NodeID` is 20 bytes which is the node ID of the validator.
 - `StartTime` is a long which is the Unix time when the validator starts validating.
 - `EndTime` is a long which is the Unix time when the validator stops validating.
 - `Weight` is a long which is the amount the validator stakes
 - **Stake** Stake has `LockedOuts`
 - `LockedOuts` An array of Transferable Outputs that are locked for the duration of the staking period. At the end of the staking period, these outputs are refunded to their respective addresses.
 - **RewardsOwner** A `SECP256K1OutputOwners`
 - **Shares** 10,000 times percentage of reward taken from delegators

Gantt Unsigned Add Validator TX Specification

```

+-----+-----+
| base_tx      : BaseTx           |           size(base_tx) bytes |
+-----+-----+
| validator    : Validator        |           44 bytes |
+-----+-----+
| stake         : Stake           |           sizeLockedOuts bytes |
+-----+-----+
| rewards_owner : SECP256K1OutputOwners |           size(rewards_owner) bytes |
+-----+-----+
| shares        : Shares          |           4 bytes |
+-----+-----+
                                         | 48 + size(stake) + size(rewards_owner) + size(base_tx) bytes |
+-----+-----+

```

Proto Unsigned Add Validator TX Specification

```
message AddValidatorTx {
    BaseTx base_tx = 1;                                // size(base_tx)
    Validator validator = 2;                            // 44 bytes
    Stake stake = 3;                                  // size(LockedOuts)
    SEC256K1OutputOwners rewards_owner = 4; // size(rewards_owner)
    uint32 shares = 5;                                 // 04 bytes
}
```

Unsigned Add Validator TX Example

Let's make an unsigned add validator TX that uses the inputs and outputs from the previous examples:

- **RewardsOwner** : 0x00000000b00000000000000000000000000000000100000001da2bee01be82ecc00c34f361eda8eb30fb5a715c
 - **Shares** : 0x00000064

```

0x00, 0x00, 0x00, 0x64,
]

```

Unsigned Remove Subnet Validator TX

What Unsigned Remove Subnet Validator TX Contains

An unsigned remove Subnet validator TX contains a `BaseTx`, `NodeID`, `SubnetID`, and `SubnetAuth`. The `TypeID` for this type is 23 or `0x00000017`.

- `BaseTx`
- `NodeID` is the 20 byte node ID of the validator.
- `SubnetID` is the 32 byte Subnet ID that the validator is being removed from.
- `SubnetAuth` contains `SigIndices` and has a type id of `0x000000a`. `SigIndices` is a list of unique ints that define the addresses signing the control signature which proves that the issuer has the right to remove the node from the Subnet. The array must be sorted low to high.

Gantt Unsigned Remove Subnet Validator TX Specification

		size(base_tx) bytes
base_tx	: BaseTx	
node_id	: [20]byte	20 bytes
subnet_id	: [32]byte	32 bytes
sig_indices	: SubnetAuth	4 bytes + len(sig_indices) bytes
56 + len(sig_indices) + size(base_tx)		bytes

Proto Unsigned Remove Subnet Validator TX Specification

```

message RemoveSubnetValidatorTx {
    BaseTx base_tx = 1;           // size(base_tx)
    string node_id = 2;          // 20 bytes
    SubnetID subnet_id = 3;      // 32 bytes
    SubnetAuth subnet_auth = 4;   // 04 bytes + len(sig_indices)
}

```

Unsigned Remove Subnet Validator TX Example

Let's make an unsigned remove Subnet validator TX that uses the inputs and outputs from the previous examples:

- `BaseTx` : "Example BaseTx as defined above with ID set to 17"
- `NodeID` : `0xe902a9a86640bfdb1cd0e36c0cc982b83e5765fa`
- `SubnetID` : `0x4a177205df5c29929d06db9d941f83d5ea985de302015e99252d16469a6610db`
- `SubnetAuth` : `0x0000000a0000000100000000`

```

[
    BaseTx     <- 0x0000000000013d0ad12b8ee8928edf248ca91ca55600fb383f07c32bfff1d6dec472b25cf59a70000000000000000000000000000000
    NodeID    <- 0xe902a9a86640bfdb1cd0e36c0cc982b83e5765fa
    SubnetID   <- 0x4a177205df5c29929d06db9d941f83d5ea985de302015e99252d16469a6610db
    SubnetAuth <- 0x0000000a0000000100000000
]
=
[
    // BaseTx
    0x00, 0x00, 0x00, 0x17, 0x00, 0x00, 0x30, 0x39,
    0x3d, 0x0a, 0xd1, 0x2b, 0x8e, 0x8e, 0x92, 0x8e,
    0xdf, 0x24, 0x8c, 0xa9, 0x1c, 0xa5, 0x56, 0x00,
    0xfb, 0x38, 0x3f, 0x07, 0xc3, 0x2b, 0xff, 0x1d,
    0x6d, 0xec, 0x47, 0x2b, 0x25, 0xcf, 0x59, 0xa7,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00,
    // NodeID
    0xe9, 0x02, 0xa9, 0xa8, 0x66, 0x40, 0xbf, 0xdb,
    0x1c, 0xd0, 0xe3, 0x6c, 0x0c, 0xc9, 0x82, 0xb8,
    0x3e, 0x57, 0x65, 0xfa,
    // SubnetID
    0x4a, 0x17, 0x72, 0x05, 0xdf, 0x5c, 0x29, 0x92,
    0x9d, 0x06, 0xdb, 0x9d, 0x94, 0x1f, 0x83, 0xd5,
    0xea, 0x98, 0x5d, 0xe3, 0x02, 0x01, 0x5e, 0x99,
    0x25, 0x2d, 0x16, 0x46, 0x9a, 0x66, 0x10,
    // SubnetAuth
    // SubnetAuth TypeID
    0x00, 0x00, 0x00, 0xa,
]
```

```

    // SigIndices length
    0x00, 0x00, 0x00, 0x01,
    // SigIndices
    0x00, 0x00, 0x00, 0x00,
]

```

Unsigned Add Permissionless Validator TX

What Unsigned Add Permissionless Validator TX Contains

An unsigned add permissionless validator TX contains a `BaseTx`, `Validator`, `SubnetID`, `Signer`, `StakeOuts`, `ValidatorRewardsOwner`, `DelegatorRewardsOwner`, and `DelegationShares`. The `TypeID` for this type is 25 or `0x00000019`.

- **BaseTx**
- **Validator** Validator has a `NodeID`, `StartTime`, `EndTime`, and `Weight`
 - `NodeID` is the 20 byte node ID of the validator.
 - `StartTime` is a long which is the Unix time when the validator starts validating.
 - `EndTime` is a long which is the Unix time when the validator stops validating.
 - `Weight` is a long which is the amount the validator stakes
- **SubnetID** is the 32 byte Subnet ID of the Subnet this validator will validate.
- **Signer** If the [SubnetID] is the primary network, [Signer] is the type ID 27 (`0x1B`) followed by a [Proof of Possession](#). If the [SubnetID] is not the primary network, this value is the empty signer, whose byte representation is only the type ID 28 (`0x1C`).
- **StakeOuts** An array of Transferable Outputs. Where to send staked tokens when done validating.
- **ValidatorRewardsOwner** Where to send validation rewards when done validating.
- **DelegatorRewardsOwner** Where to send delegation rewards when done validating.
- **DelegationShares** a short which is the fee this validator charges delegators as a percentage, times 10,000. For example, if this validator has `DelegationShares=300,000` then they take 30% of rewards from delegators.

Gantt Unsigned Add Permissionless Validator TX Specification

base_tx	: BaseTx	size(base_tx) bytes
validator	: Validator	44 bytes
subnet_id	: [32]byte	32 bytes
signer	: Signer	144 bytes
stake_outs	: []TransferOut	4 + size(stake_outs) bytes
validator_rewards_owner	: SECP256K1OutputOwners	size(validator_rewards_owner) bytes
delegator_rewards_owner	: SECP256K1OutputOwners	size(delegator_rewards_owner) bytes
delegation_shares	: uint32	4 bytes
		232 + size(base_tx) + size(stake_outs) +
		size(validator_rewards_owner) + size(delegator_rewards_owner) bytes

Proto Unsigned Add Permissionless Validator TX Specification

```

message AddPermissionlessValidatorTx {
    BaseTx base_tx = 1;           // size(base_tx)
    Validator validator = 2;      // 44 bytes
    SubnetID subnet_id = 3;       // 32 bytes
    Signer signer = 4;           // 148 bytes
    repeated TransferOut stake_outs = 5; // 4 bytes + size(stake_outs)
    SECP256K1OutputOwners validator_rewards_owner = 6; // size(validator_rewards_owner) bytes
    SECP256K1OutputOwners delegator_rewards_owner = 7; // size(delegator_rewards_owner) bytes
    uint32 delegation_shares = 8; // 4 bytes
}

```

Unsigned Add Permissionless Validator TX Example

Let's make an unsigned add permissionless validator TX that uses the inputs and outputs from the previous examples:

- **BaseTx** : "Example BaseTx as defined above with ID set to 1a"
- **Validator** : `0x5fa29ed4356903dac2364713c60f57d8472c7dda00000006397616e000000063beee6e000001d1a94a2000`
- **SubnetID** : `0xf3086d7bfc35be1c68db664ba9ce61a2060126b0d6b4bfb09fd7a5fb7678cada`
- **Signer** :
`0x0000001ca5af179e4188583893c2b99e1a8be27d90a9213cfbff1d75b74fe2bc9f3b072c2ded0863a9d9acd9033f223295810e429238e28d3c9b7f7212b63c`


```

0xb6, 0x3f, 0xf1, 0xee, 0x06, 0x8f, 0xbf, 0xc2,
0x4, 0xc8, 0xcd, 0x2d, 0x08, 0xeb, 0xf2, 0x97,
// Stake outs
// Num stake outs
0x00, 0x00, 0x00, 0x01,
// AssetID
0x3d, 0x0a, 0xd1, 0x2b, 0x8e, 0xe8, 0x92, 0x8e,
0xdf, 0x24, 0x8c, 0xa9, 0x1c, 0xa5, 0x56, 0x00,
0xfb, 0x38, 0x3f, 0x07, 0xc3, 0x2b, 0xff, 0x1d,
0x6d, 0xec, 0x47, 0x2b, 0x25, 0xcf, 0x59, 0xa7,
// Output
// typeID
0x00, 0x00, 0x00, 0x07,
// Amount
0x00, 0x00, 0x01, 0xd1, 0xa9, 0x4a, 0x20, 0x00,
// Locktime
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// Threshold
0x00, 0x00, 0x00, 0x01,
// Num addrs
0x00, 0x00, 0x00, 0x01,
// Addr 0
0x33, 0xee, 0xff, 0xc6, 0x47, 0x85, 0xcf, 0x9d,
0x80, 0xe7, 0x73, 0x1d, 0x9f, 0x31, 0xf6, 0x7b,
0xd0, 0x3c, 0x5c, 0xf0,
// Validator rewards owner
// TypeID
0x00, 0x00, 0x00, 0x0b,
// Locktime
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// Threshold
0x00, 0x00, 0x00, 0x01,
// Num addrs
0x00, 0x00, 0x00, 0x01,
// Addr 0
0x72, 0xf3, 0xeb, 0x9a, 0xea, 0xf8, 0x28, 0x30,
0x11, 0xce, 0x6e, 0x43, 0x7f, 0xde, 0xcd, 0x65,
0xea, 0xce, 0x8f, 0x52,
// Delegator rewards owner
// TypeID
0x00, 0x00, 0x00, 0x0b,
// Locktime
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// Threshold
0x00, 0x00, 0x00, 0x01,
// Num addrs
0x00, 0x00, 0x00, 0x01,
// Addr 0
0xb2, 0xb9, 0x13, 0x13, 0xac, 0x48, 0x7c, 0x22,
0x24, 0x45, 0x25, 0x4e, 0x26, 0xcd, 0x02, 0x6d,
0x21, 0xf6, 0xf4, 0x40,
// Delegation shares
0x00, 0x00, 0x4e, 0x20,
]

```

Unsigned Add Permissionless Delegator TX

What Unsigned Add Permissionless Delegator TX Contains

An unsigned add permissionless delegator TX contains a `BaseTx`, `Validator`, `SubnetID`, `StakeOuts`, and `DelegatorRewardsOwner`. The `TypeID` for this type is 26 or `0x0000001a`.

- `BaseTx`
- `Validator`: Validator has a `NodeID`, `StartTime`, `EndTime`, and `Weight`
 - `NodeID` is the 20 byte node ID of the validator.
 - `StartTime` is a long which is the Unix time when the validator starts validating.
 - `EndTime` is a long which is the Unix time when the validator stops validating.
 - `Weight` is a long which is the amount the validator stakes
- `SubnetID` is the 32 byte Subnet ID of the Subnet this delegation is on.
- `StakeOuts`: An array of Transferable Outputs. Where to send staked tokens when done validating.
- `DelegatorRewardsOwner`: Where to send staking rewards when done validating.

Gantt Unsigned Add Permissionless Delegator TX Specification

<code>base_tx</code>	<code>: BaseTx</code>	<code>size(base_tx) bytes</code>

```
+-----+-----+
| validator      : Validator          |           44 bytes |
+-----+-----+
| subnet_id     : [32]byte           |           32 bytes |
+-----+-----+
| stake_outs    : []TransferOut      |           4 + size(stake_outs) bytes |
+-----+-----+
| delegator_rewards_owner : SECP256K1OutputOwners | size(delegator_rewards_owner) bytes |
+-----+-----+
| 80 + size(base_tx) + size(stake_outs) + size(delegator_rewards_owner) bytes |
+-----+-----+
```

Proto Unsigned Add Permissionless Delegator TX Specification

```
message AddPermissionlessDelegatorTx {
    BaseTx base_tx = 1;           // size(base_tx)
    Validator validator = 2;     // size(validator)
    SubnetID subnet_id = 3;      // 32 bytes
    repeated TransferOut stake_outs = 4; // 4 bytes + size(stake_outs)
    SECP256K1OutputOwners delegator_rewards_owner = 5; // size(delegator_rewards_owner) bytes
}
```

Unsigned Add Permissionless Delegator TX Example

Let's make an unsigned add permissionless delegator TX that uses the inputs and outputs from the previous examples:

```

0x00, 0x00, 0x00, 0x01,
// Stake out 0
// AssetID
0x3d, 0x0a, 0xd1, 0x2b, 0x8e, 0xe8, 0x92, 0x8e,
0xdf, 0x24, 0x8c, 0xa9, 0x1c, 0xa5, 0x56, 0x00,
0xfb, 0x38, 0x3f, 0x07, 0xc3, 0x2b, 0xff, 0x1d,
0x6d, 0xec, 0x47, 0x2b, 0x25, 0xcf, 0x59, 0xa7,
// TypeID
0x00, 0x00, 0x00, 0x07,
// Amount
0x00, 0x00, 0x01, 0xd1, 0xa9, 0x4a, 0x20, 0x00,
// Locktime
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// Threshold
0x00, 0x00, 0x00, 0x01,
// Num addrs
0x00, 0x00, 0x00, 0x01,
// Addr 0
0x33, 0xee, 0xff, 0xc6, 0x47, 0x85, 0xcf, 0x9d,
0x80, 0x7, 0x73, 0xd, 0x9f, 0x31, 0xf6, 0x7b,
0xd0, 0x3c, 0x5c, 0xf0,
// Delegator_rewards_owner
// TypeID
0x00, 0x00, 0x00, 0xb,
// Locktime
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// Threshold
0x00, 0x00, 0x00, 0x01,
// Num addrs
0x00, 0x00, 0x00, 0x01,
// Addr 0
0x72, 0xf3, 0xeb, 0x9a, 0xea, 0xf8, 0x28, 0x30,
0x11, 0xce, 0x6e, 0x43, 0x7f, 0xde, 0xcd, 0x65,
0xea, 0xce, 0x8f, 0x52,
]

```

Unsigned Transform Subnet TX

Transforms a permissioned Subnet into a permissionless Subnet. Must be signed by the Subnet owner.

What Unsigned Transform Subnet TX Contains

An unsigned transform Subnet TX contains a `BaseTx`, `SubnetID`, `AssetID`, `InitialSupply`, `MaximumSupply`, `MinConsumptionRate`, `MaxConsumptionRate`, `MinValidatorStake`, `MaxValidatorStake`, `MinStakeDuration`, `MaxStakeDuration`, `MinDelegationFee`, `MinDelegatorStake`, `MaxValidatorWeightFactor`, `UptimeRequirement`, and `SubnetAuth`. The `TypeID` for this type is 24 or `0x00000018`.

- **BaseTx**
- **SubnetID** a 32-byte Subnet ID of the Subnet to transform.
- **AssetID** is a 32-byte array that defines which asset to use when staking on the Subnet.
 - Restrictions
 - Must not be the Empty ID
 - Must not be the AVAX ID
- **InitialSupply** is a long which is the amount to initially specify as the current supply.
 - Restrictions
 - Must be > 0
- **MaximumSupply** is a long which is the amount to specify as the maximum token supply.
 - Restrictions
 - Must be \geq [InitialSupply]
- **MinConsumptionRate** is a long which is the rate to allocate funds if the validator's stake duration is 0.
- **MaxConsumptionRate** is a long which is the rate to allocate funds if the validator's stake duration is equal to the minting period.
 - Restrictions
 - Must be \geq [MinConsumptionRate]
 - Must be \leq [reward.PercentDenominator]
- **MinValidatorStake** is a long which the minimum amount of funds required to become a validator.
 - Restrictions
 - Must be > 0
 - Must be \leq [InitialSupply]
- **MaxValidatorStake** is a long which is the maximum amount of funds a single validator can be allocated, including delegated funds.
 - Restrictions:
 - Must be \geq [MinValidatorStake]
 - Must be \leq [MaximumSupply]
- **MinStakeDuration** is a short which is the minimum number of seconds a staker can stake for.

- Restrictions
 - Must be > 0
- **MaxStakeDuration** is a short which is the maximum number of seconds a staker can stake for.
 - Restrictions
 - Must be \geq [MinStakeDuration]
 - Must be \leq [GlobalMaxStakeDuration]
- **MinDelegationFee** is a short is the minimum percentage a validator must charge a delegator for delegating.
 - Restrictions
 - Must be \leq [reward.PercentDenominator]
- **MinDelegatorStake** is a short which is the minimum amount of funds required to become a delegator.
 - Restrictions
 - Must be > 0
- **MaxValidatorWeightFactor** is a byte which is the factor which calculates the maximum amount of delegation a validator can receive. Note: a value of 1 effectively disables delegation.
 - Restrictions
 - Must be > 0
- **UptimeRequirement** is a short which is the minimum percentage a validator must be online and responsive to receive a reward.
 - Restrictions
 - Must be \leq [reward.PercentDenominator]
- **SubnetAuth** contains **SigIndices** and has a type id of `0x0000000a`. **SigIndices** is a list of unique ints that define the addresses signing the control signature to authorizes this transformation. The array must be sorted low to high.

Gantt Unsigned Transform Subnet TX Specification

base_tx : BaseTx	size(base_tx) bytes
subnet_id : [32]byte	32 bytes
asset_id : [32]byte	32 bytes
initial_supply : long	8 bytes
maximum_supply : long	8 bytes
min_consumption_rate : long	8 bytes
max_consumption_rate : long	8 bytes
min_validator_stake : long	8 bytes
max_validator_stake : long	8 bytes
min_stake_duration : short	4 bytes
max_stake_duration : short	4 bytes
min_delegation_fee : short	4 bytes
min_delegator_stake : long	8 bytes
max_validator_weight_factor : byte	1 byte
uptime_requirement : short	4 bytes
subnet_auth : SubnetAuth	4 bytes + len(sig_indices) bytes
141 + size(base_tx) + len(sig_indices) bytes	

Proto Unsigned Transform Subnet TX Specification

```
message TransformSubnetTx {
    BaseTx base_tx = 1;           // size(base_tx)
    SubnetID subnet_id = 2;       // 32 bytes
    bytes asset_id = 3;           // 32 bytes
    uint64 initial_supply = 4;    // 08 bytes
    uint64 maximum_supply = 5;    // 08 bytes
    uint64 min_consumption_rate = 6; // 08 bytes
    uint64 max_consumption_rate = 7; // 08 bytes
    uint64 min_validator_stake = 8; // 08 bytes
```

```

        uint64 max_validator_stake = 9; // 08 bytes
        uint32 min_stake_duration = 10; // 04 bytes
        uint32 max_stake_duration = 11; // 04 bytes
        uint32 min_delegation_fee = 12; // 04 bytes
        uint32 min_delegator_stake = 13; // 08 bytes
        byte max_validator_weight_factor = 14; // 01 byte
        uint32 uptime_requirement = 15; // 04 bytes
        SubnetAuth subnet_auth = 16; // 04 bytes + len(sig_indices)
    }
}

```

Unsigned Transform Subnet TX Example

Let's make an unsigned transform Subnet TX that uses the inputs and outputs from the previous examples:

- **BaseTx** : "Example BaseTx as defined above with ID set to 18"
- **SubnetID** : 0x5fa29ed4356903dac2364713c60f57d8472c7dda4a5e08d88a88ad8ea71aed60
- **AssetID** : 0xf3086d7bfc35be1c68db664ba9ce61a2060126b0d6b4bfb09fd7a5fb7678cada
- **InitialSupply** : 0x000000e8d4a51000
- **MaximumSupply** : 0x000009184e72a000
- **MinConsumptionRate** : 0x00000000000000001
- **MaxConsumptionRate** : 0x0000000000000000a
- **MinValidatorStake** : 0x000000174876e800
- **MaxValidatorStake** : 0x000001dia94a2000
- **MinStakeDuration** : 0x00015180
- **MaxStakeDuration** : 0x01e13380
- **MinDelegationFee** : 0x00002710
- **MinDelegatorStake** : 0x000000174876e800
- **MaxValidatorWeightFactor** : 0x05
- **UptimeRequirement** : 0x000c3500
- **SubnetAuth** :
 - **TypeID** : 0x0000000a
 - **SigIndices** : 0x00000000

```

[
  BaseTx      <-
0000001800003039e902a9a86640bfd81cd0e36c0cc982b83e5765fad5f6bbe6abdcce7b5ae7d7c7000000000000000014a177205df5c29929d06db9d941f83d5ea<
  SubnetID    <- 0x5fa29ed4356903dac2364713c60f57d8472c7dda4a5e08d88a88ad8ea71aed60
  AssetID     <- 0xf3086d7bfc35be1c68db664ba9ce61a2060126b0d6b4bfb09fd7a5fb7678cada
  InitialSupply <- 0x000000e8d4a51000
  MaximumSupply <- 0x000009184e72a000
  MinConsumptionRate <- 0x00000000000000001
  MaxConsumptionRate <- 0x0000000000000000a
  MinValidatorStake <- 0x000000174876e800
  MaxValidatorStake <- 0x000001dia94a2000
  MinStakeDuration <- 0x00015180
  MaxStakeDuration <- 0x01e13380
  MinDelegationFee <- 0x00002710
  MinDelegatorStake <- 0x000000174876e800
  MaxValidatorWeightFactor <- 0x05
  UptimeRequirement <- 0x000c3500
  SubnetAuth   <- 0x0000000a0000000100000000
]
=
[
  // BaseTx:
  0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x30, 0x39,
  0xe9, 0x02, 0xa9, 0xa8, 0x66, 0x40, 0xbff, 0xdb,
  0x1c, 0xd0, 0xe3, 0x6c, 0x0c, 0xc9, 0x82, 0xb8,
  0x3e, 0x57, 0x65, 0xfa, 0xd5, 0xff, 0xbb, 0xe6,
  0xab, 0xdc, 0xce, 0x7b, 0x5a, 0xe7, 0xd7, 0xc7,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
  0x4a, 0x17, 0x72, 0x05, 0xdf, 0x5c, 0x29, 0x92,
  0x9d, 0x06, 0xdb, 0xd, 0x94, 0x1f, 0x83, 0xd5,
  0xea, 0x98, 0x5d, 0xe3, 0x02, 0x01, 0x5e, 0x99,
  0x25, 0x2d, 0x16, 0x46, 0x9a, 0x66, 0x10, 0x2b,
  0x00, 0x00, 0x00, 0x00, 0x3d, 0xa, 0xd1, 0x2b,
  0x8e, 0xe8, 0x92, 0x8e, 0xdf, 0x24, 0x8c, 0xa9,
  0x1c, 0xa5, 0x56, 0x00, 0xfb, 0x38, 0x3f, 0x07,
  0xc3, 0x2b, 0xff, 0x1d, 0x6d, 0xec, 0x47, 0x2b,
  0x25, 0xcf, 0x59, 0xa7, 0x00, 0x00, 0x00, 0x05,
  0x00, 0x00, 0x00, 0x00, 0x0f, 0x42, 0x40,
  0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x5f, 0xa2, 0x9e, 0xd4,
  0x35, 0x69, 0x03, 0xda, 0xc2, 0x36, 0x47, 0x13,
  0xc6, 0x0f, 0x57, 0xd8, 0x47, 0x2c, 0x7d, 0xda,
]
```

```

0x4a, 0x5e, 0x08, 0xd8, 0x8a, 0x88, 0xad, 0x8e,
0xa7, 0x1a, 0xed, 0x60, 0xf3, 0x08, 0x6d, 0x7b,
0xfc, 0x35, 0xbe, 0x1c, 0x68, 0xdb, 0x66, 0x4b,
0xa9, 0xce, 0x61, 0xa2, 0x06, 0x01, 0x26, 0xb0,
0xd6, 0xb4, 0xbf, 0xb0, 0x9f, 0xd7, 0xa5, 0xfb,
0x76, 0x78, 0xca, 0xda, 0x00, 0x00, 0xe8,
0xd4, 0xa5, 0x10, 0x00, 0x00, 0x00, 0x09, 0x18,
0x4e, 0x72, 0xa0, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xa0, 0x00, 0x00, 0x00, 0x17,
0x48, 0x76, 0xe8, 0x00, 0x00, 0x01, 0xd1,
0xa9, 0x4a, 0x20, 0x00, 0x00, 0x01, 0x51, 0x80,
0x01, 0xe1, 0x33, 0x80, 0x00, 0x00, 0x27, 0x10,
0x00, 0x00, 0x00, 0x17, 0x48, 0x76, 0xe8, 0x00,
0x05, 0x00, 0xc, 0x35, 0x00, 0x00, 0x00, 0x00,
0xa, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00,
0x00,
// SubnetID
0x5f, 0xa2, 0x9e, 0xd4, 0x35, 0x69, 0x03, 0xda,
0xc2, 0x36, 0x47, 0x13, 0xc6, 0x0f, 0x57, 0xd8,
0x47, 0x2c, 0x7d, 0xda, 0x4a, 0x5e, 0x08, 0xd8,
0x8a, 0x88, 0xad, 0x8e, 0xa7, 0x1a, 0xed, 0x60,
// AssetID
0xf3, 0x08, 0x6d, 0x7b, 0xfc, 0x35, 0xbe, 0x1c,
0x8, 0xdb, 0x66, 0x4b, 0xa9, 0xce, 0x61, 0xa2,
0x06, 0x01, 0x26, 0xb0, 0xd6, 0xb4, 0xbf, 0xb0,
0x9f, 0xd7, 0xa5, 0xfb, 0x76, 0x78, 0xca, 0xda,
// InitialSupply
0x00, 0x00, 0x00, 0x08, 0x18, 0x4e, 0x72, 0xa0, 0x00,
// MaximumSupply
0x00, 0x00, 0x09, 0x18, 0x4e, 0x72, 0xa0, 0x00,
// MinConsumptionRate
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
// MaxConsumptionRate
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xa,
// MinValidatorStake
0x00, 0x00, 0x00, 0x17, 0x48, 0x76, 0xe8, 0x00,
// MaxValidatorStake
0x00, 0x00, 0x01, 0xd1, 0xa9, 0x4a, 0x20, 0x00,
// MinStakeDuration
0x00, 0x01, 0x51, 0x80,
// MaxStakeDuration
0x01, 0xe1, 0x33, 0x80,
// MinDelegationFee
0x00, 0x00, 0x27, 0x10,
// MinDelegatorStake
0x00, 0x00, 0x00, 0x17, 0x48, 0x76, 0xe8, 0x00,
// MaxValidatorWeightFactor
0x05,
// UptimeRequirement
0x00, 0x0c, 0x35, 0x00,
// SubnetAuth
// SubnetAuth TypeID
0x00, 0x00, 0x00, 0xa,
// SigIndices length
0x00, 0x00, 0x00, 0x01,
// SigIndices
0x00, 0x00, 0x00, 0x00,
]

```

Unsigned Add Subnet Validator TX

What Unsigned Add Subnet Validator TX Contains

An unsigned add Subnet validator TX contains a `BaseTx`, `Validator`, `SubnetID`, and `SubnetAuth`. The `TypeID` for this type is `0x0000000d`.

- **BaseTx**
- **Validator** Validator has a `NodeID`, `StartTime`, `EndTime`, and `Weight`
 - `NodeID` is the 20 byte node ID of the validator.
 - `StartTime` is a long which is the Unix time when the validator starts validating.
 - `EndTime` is a long which is the Unix time when the validator stops validating.
 - `Weight` is a long which is the amount the validator stakes
- **SubnetID** is the 32 byte Subnet ID to add the validator to.
- **SubnetAuth** contains `SigIndices` and has a type id of `0x000000a`. `SigIndices` is a list of unique ints that define the addresses signing the control signature to add a validator to a Subnet. The array must be sorted low to high.

Gantt Unsigned Add Subnet Validator TX Specification

```

+-----+-----+
| base_tx      : BaseTx           |          size(base_tx) bytes |
+-----+-----+
| validator    : Validator        |          44 bytes |
+-----+-----+
| subnet_id    : [32]byte         |          32 bytes |
+-----+-----+
| subnet_auth  : SubnetAuth       |          4 bytes + len(sig_indices) bytes |
+-----+-----+
                                         | 80 + len(sig_indices) + size(base_tx) bytes |
+-----+-----+

```

Proto Unsigned Add Subnet Validator TX Specification

```
message AddSubnetValidatorTx {
    BaseTx base_tx = 1;           // size(base_tx)
    Validator validator = 2;      // size(validator)
    SubnetID subnet_id = 3;       // 32 bytes
    SubnetAuth subnet_auth = 4;   // 04 bytes + len(sig_indices)
}
```

Unsigned Add Subnet Validator TX Example

Let's make an unsigned add Subnet validator TX that uses the inputs and outputs from the previous examples:

- **BaseTx** : "Example BaseTx as defined above with ID set to 0d"
 - **NodeID** : 0xe9094f73698002fd52c90819b457b9fbc866ab80
 - **StartTime** : 0x000000005f21f31d
 - **EndTime** : 0x000000005f497dc6
 - **Weight** : 0x000000000000a431
 - **SubnetID** : 0x58b1092871db85bc752742054e2e8be0adf8166ec1f0f0769f4779f14c71d7eb
 - **SubnetAuth** :
 - **TypeID** : 0x0000000a
 - **SigIndices** : 0x00000000

```

0x68, 0x70, 0xb7, 0xd6, 0x6a, 0xc3, 0x25, 0x40,
0x31, 0x13, 0x79, 0xe5, 0xb5, 0xdb, 0xad, 0x28,
0xec, 0x7e, 0xb8, 0xdd, 0xbf, 0xc8, 0xf4, 0xd6,
0x72, 0x99, 0xeb, 0xb4, 0x84, 0x75, 0x90, 0x7a,
0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
0xee, 0x6b, 0x28, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00,
// Node ID
0xe9, 0x09, 0x4f, 0x73, 0x69, 0x80, 0x02, 0xfd,
0x52, 0xc9, 0x08, 0x19, 0xb4, 0x57, 0xb9, 0xfb,
0xc8, 0x66, 0xab, 0x80,
// StartTime
0x00, 0x00, 0x00, 0x00, 0x5f, 0x21, 0xf3, 0x1d,
// EndTime
0x00, 0x00, 0x00, 0x00, 0x5f, 0x49, 0x7d, 0xc6,
// Weight
0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
// SubnetID
0x58, 0xb1, 0x09, 0x28, 0x71, 0xdb, 0x85, 0xbc,
0x75, 0x27, 0x42, 0x05, 0x4e, 0x2e, 0xb8, 0xe0,
0xad, 0xf8, 0x16, 0x6e, 0xc1, 0xf0, 0xf0, 0x76,
0x9f, 0x47, 0x79, 0xf1, 0x4c, 0x71, 0xd7, 0xeb,
// SubnetAuth
// SubnetAuth TypeID
0x00, 0x00, 0x00, 0xa,
// SigIndices length
0x00, 0x00, 0x00, 0x01,
// SigIndices
0x00, 0x00, 0x00, 0x00,
]

```

Unsigned Add Delegator TX

What Unsigned Add Delegator TX Contains

An unsigned add delegator TX contains a `BaseTx`, `Validator`, `Stake`, and `RewardsOwner`. The `TypeID` for this type is `0x0000000e`.

- **BaseTx**
- **Validator** Validator has a `NodeID`, `StartTime`, `EndTime`, and `Weight`
 - `NodeID` is 20 bytes which is the node ID of the delegatee.
 - `StartTime` is a long which is the Unix time when the delegator starts delegating.
 - `EndTime` is a long which is the Unix time when the delegator stops delegating (and staked AVAX is returned).
 - `Weight` is a long which is the amount the delegator stakes
- **Stake** Stake has `LockedOuts`
 - `LockedOuts` An array of Transferable Outputs that are locked for the duration of the staking period. At the end of the staking period, these outputs are refunded to their respective addresses.
- **RewardsOwner** An `SECP256K1OutputOwners`

Gantt Unsigned Add Delegator TX Specification

<code>base_tx</code>	<code>: BaseTx</code>	<code>size(base_tx)</code> bytes
<code>validator</code>	<code>: Validator</code>	44 bytes
<code>stake</code>	<code>: Stake</code>	<code>size(LockedOuts)</code> bytes
<code>rewards_owner</code>	<code>: SECP256K1OutputOwners</code>	<code>size(rewards_owner)</code> bytes
		<code>44 + size(stake) + size(rewards_owner) + size(base_tx)</code> bytes

Proto Unsigned Add Delegator TX Specification

```

message AddDelegatorTx {
    BaseTx base_tx = 1; // size(base_tx)
    Validator validator = 2; // 44 bytes
    Stake stake = 3; // size(LockedOuts)
    SECP256K1OutputOwners rewards_owner = 4; // size(rewards_owner)
}

```

Unsigned Add Delegator TX Example

Let's make an unsigned add delegator TX that uses the inputs and outputs from the previous examples

```
// RewardsOwner  
0x00, 0x00, 0x00, 0xb0, 0x00, 0x00, 0x00, 0x00,  
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,  
0x00, 0x00, 0x00, 0x01, 0xda, 0x2b, 0xee, 0x01,  
0xbe, 0x82, 0xec, 0xc0, 0x0c, 0x34, 0xf3, 0x61,  
0xed, 0xa8, 0xeb, 0x30, 0xfb, 0x5a, 0x71, 0x5c,
```

Unsigned Create Chain TX

What Unsigned Create Chain TX Contains

An unsigned create chain TX contains a `BaseTx`, `SubnetID`, `ChainName`, `VMID`, `FxIDs`, `GenesisData` and `SubnetAuth`. The `TypeID` for this type is `0x0000000f`.

- **BaseTx**
 - **SubnetID** ID of the Subnet that validates this blockchain
 - **ChainName** A human readable name for the chain; need not be unique
 - **VMID** ID of the VM running on the new chain
 - **FxIDs** IDs of the feature extensions running on the new chain
 - **GenesisData** Byte representation of genesis state of the new chain
 - **SubnetAuth** Authorizes this blockchain to be added to this Subnet

Gantt Unsigned Create Chain TX Specification

```

+-----+-----+
| base_tx      : BaseTx          |           size(base_tx) bytes |
+-----+-----+
| subnet_id    : SubnetID        |           32 bytes |
+-----+-----+
| chain_name   : ChainName       |           2 + len(chain_name) bytes |
+-----+-----+
| vm_id        : VMID           |           32 bytes |
+-----+-----+
| fx_ids       : FxIDs          |           4 + size(fx_ids) bytes |
+-----+-----+
| genesis_data : GenesisData    |           4 + size(genesis_data) bytes |
+-----+-----+
| subnet_auth  : SubnetAuth     |           size(subnet_auth) bytes |
+-----+-----+
                                         | 74 + size(base_tx) + size(chain_name) + size(fx_ids) + |
                                         |           size(genesis_data) + size(subnet_auth) bytes |
+-----+-----+

```

Proto Unsigned Create Chain TX Specification

```
message CreateChainTx {
    BaseTx base_tx = 1;           // size(base_tx)
    SubnetID subnet_id = 2;       // 32 bytes
    ChainName chain_name = 3;     // 2 + len(chain_name) bytes
    VMID vmid_id = 4;            // 32 bytes
    FxIDs fx_ids = 5;            // 4 + size(fx_ids) bytes
    GenesisData genesis_data = 6 // 4 + size(genesis_data) bytes
    SubnetAuth subnet_auth = 7;    // size(subnet_auth) bytes
}
```

Unsigned Create Chain TX Example

Let's make an unsigned create chain TX that uses the inputs and outputs from the previous examples.

- **BaseTx** : "Example BaseTx as defined above with ID set to 0f"
 - **SubnetID** : 24tZhrm8j8GCJRE9PomW8FaeqbgGS4UAQjJnqqn8pq5NwYSYV1
 - **ChainName** : EPIC AVM
 - **VMID** : avm
 - **FxIDs** : [secp256k1fx]
 - **GenesisData** :
11111DdZMhYXUiFV9FNpfptSroystsXzbWicG954YAKfkkrk3bCEzLVi7gunleAmAwMiQzVhtGpdR6dnPVcfhBE7brzkJ1r4wzi3dgA8G9Jwc4WpZ6Uh4Dr9aTdw7sFA!
• **SubmitAuth** : 0x000000000000000100000000


```
0xe2, 0x8c, 0xee, 0x6a, 0x0e, 0xbd, 0x09, 0xf1,  
0xfe, 0x88, 0x4f, 0x68, 0x61, 0xe1, 0xb2, 0x9c,  
  
// type id (Subnet Auth)  
0x00, 0x00, 0x00, 0x0a,  
// num address indices  
0x00, 0x00, 0x00, 0x01,  
// address index  
0x00, 0x00, 0x00, 0x00,  
1
```

Unsigned Create Subnet TX

What Unsigned Create Subnet TX Contains

An unsigned create Subnet TX contains a `BaseTx`, and `RewardsOwner`. The `TypeID` for this type is `0x000000010`

- **BaseTx**
 - **RewardsOwner** A SECP256K1OutputOwners

Gantt Unsigned Create Subnet TX Specification

```

+-----+-----+
| base_tx      : BaseTx           |      size(base_tx) bytes |
+-----+-----+
| rewards_owner : SECP256K1OutputOwners |      size(rewards_owner) bytes |
+-----+-----+
                                         | size(rewards_owner) + size(base_tx) bytes |
+-----+-----+

```

Proto Unsigned Create Subnet TX Specification

```
message CreateSubnetTx {
    BaseTx base_tx = 1;                                // size(base_tx)
    SECP256K1OutputOwners rewards_owner = 2; // size(rewards_owner)
}
```

Unsigned Create Subnet TX Example

Let's make an unsigned create Subnet TX that uses the inputs from the previous examples

- **BaseTx** : "Example BaseTx as defined above but with TypeID set to 16"
 - **RewardsOwner** :
 - **TypeID** : 11
 - **Locktime** : 0
 - **Threshold** : 1
 - **Addressess** : [0xda2bee01be82ecc00c34f361eda8eb30fb5a715c]

```
0x00, 0x00, 0x00, 0x01, 0x3c, 0xb7, 0xd3, 0x84,
0x2e, 0x8c, 0xee, 0x6a, 0x0e, 0xbd, 0x09, 0xff,
0xfe, 0x88, 0x4f, 0x68, 0x61, 0xe1, 0xb2, 0x9c,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// RewardsOwner type id
0x00, 0x00, 0x00, 0x0b,
// locktime:
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// threshold:
0x00, 0x00, 0x00, 0x01,
// number of addresses:
0x00, 0x00, 0x00, 0x01,
// addrs[0]:
0xda, 0x2b, 0xee, 0x01,
0xbe, 0x82, 0xec, 0xc0, 0x0c, 0x34, 0xf3, 0x61,
0xed, 0xa8, 0xeb, 0x30, 0xfb, 0x5a, 0x71, 0x5c
```

Unsigned Import TX

What Unsigned Import TX Contains

An unsigned import TX contains a `BaseTx`, `SourceChain`, and `Ins`. The `TypeID` for this type is `0x00000011`.

- **BaseTx**
 - **SourceChain** is a 32-byte source blockchain ID.
 - **Ins** is a variable length array of Transferable Inputs.

Gantt Unsigned Import TX Specification

```

+-----+-----+
| base_tx      : BaseTx      |           size(base_tx) bytes |
+-----+
| source_chain : [32]bytete   |           32 bytes |
+-----+
| ins          : []TransferIn |        4 + size(ins) bytes |
+-----+
                                         | 36 + size(ins) + size(base_tx) bytes |
+-----+

```

Proto Unsigned Import TX Specification

```
message ImportTx {
    BaseTx base_tx = 1;           // size(base_tx)
    bytes source_chain = 2;       // 32 bytes
    repeated TransferIn ins = 3; // 4 bytes + size(ins)
}
```

Unsigned Import TX Example

Let's make an unsigned import TX that uses the inputs from the previous examples:

- **BaseTx** : "Example BaseTx as defined above with TypeID set to 17"
 - **SourceChain** :
 - **Ins** : "Example SECP256K1 Transfer Input as defined above"

```

0x0d, 0xbf, 0x2d, 0x18, 0xb2, 0xd7, 0xd1, 0x68,
0x52, 0x44, 0x40, 0x55, 0x5d, 0x55, 0x00, 0x88,
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x12, 0x30,
0x9c, 0xd5, 0xfd, 0xc0, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x01, 0x3c, 0xb7, 0xd3, 0x84,
0x2e, 0x8c, 0xee, 0x6a, 0x0e, 0xbd, 0x09, 0xf1,
0xfe, 0x88, 0x4f, 0x68, 0x61, 0xe1, 0xb2, 0x9c,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
// sourceChain
0x78, 0x7c, 0xd3, 0x24, 0x3c, 0x00, 0x2e, 0x9b,
0xf5, 0xbb, 0xba, 0xea, 0x8a, 0x42, 0xa1, 0x6c,
0x1a, 0x19, 0xcc, 0x10, 0x50, 0x47, 0xc6, 0x69,
0x96, 0x80, 0x7c, 0xbef, 0x16, 0xac, 0xee, 0x10,
// input count:
0x00, 0x00, 0x00, 0x01,
// txID:
0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
// utxoIndex:
0x00, 0x00, 0x00, 0x05,
// assetID:
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
// input:
0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
0xee, 0x6b, 0x28, 0x00, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x00,
]
```

Unsigned Export TX

What Unsigned Export TX Contains

An unsigned export TX contains a `BaseTx`, `DestinationChain`, and `Outs`. The `TypeID` for this type is `0x00000012`.

- **DestinationChain** is the 32 byte ID of the chain where the funds are being exported to.
 - **Outs** is a variable length array of Transferable Outputs.

Gantt Unsigned Export TX Specification

```

+-----+-----+
| base_tx      : BaseTx      |           size(base_tx) bytes |
+-----+-----+
| destination_chain : [32]byte   |           32 bytes |
+-----+-----+
| outs          : []TransferOut |           4 + size(outs) bytes |
+-----+-----+
|                               | 36 + size(outs) + size(base_tx) bytes |
+-----+-----+

```

Proto Unsigned Export TX Specification

```
message ExportTx {
    BaseTx base_tx = 1;           // size(base_tx)
    bytes destination_chain = 2; // 32 bytes
    repeated TransferOut outs = 3; // 4 bytes + size(outs)
}
```

Unsigned Export TX Example

Let's make an unsigned export TX that uses the outputs from the previous examples:

```

Outs <- [
    0x00102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f0000000700000000000003039000000000000d4310000001000000251025c61fb<
]
=
[
    // base tx:
    0x00, 0x00, 0x00, 0x12
    0x00, 0x00, 0x00, 0x04, 0xff, 0xff, 0xff, 0xff,
    0xee, 0xee, 0xee, 0xee, 0xdd, 0xdd, 0xdd, 0xdd,
    0xcc, 0xcc, 0xcc, 0xcc, 0xbb, 0xbb, 0xbb, 0xbb,
    0xaa, 0xaa, 0xaa, 0xaa, 0x99, 0x99, 0x99, 0x99,
    0x88, 0x88, 0x88, 0x88, 0x00, 0x00, 0x00, 0x00, 0x01,
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
    0x00, 0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
    0xFB, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
    0x34, 0xbe, 0x6d, 0x2, 0x6d, 0x55, 0xa9, 0x55,
    0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
    0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,
    0x43, 0xab, 0x08, 0x59, 0x00, 0x00, 0x00, 0x01,
    0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
    0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
    0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
    0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
    0x00, 0x00, 0x00, 0x05, 0x00, 0x01, 0x02, 0x03,
    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
    0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
    0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
    0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x00, 0x05,
    0x00, 0x00, 0x00, 0x00, 0x07, 0x5b, 0xcd, 0x15,
    0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x07,
    0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x04,
    0x00, 0x01, 0x02, 0x03
    // destination_chain:
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    // outs[] count:
    0x00, 0x00, 0x00, 0x01,
    // assetID:
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0xa, 0xb, 0xc, 0xd, 0xe, 0xf,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    // output:
    0x00, 0x00, 0x00, 0x07,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x39,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
    0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x02,
    0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
    0xf6, 0x93, 0x34, 0xf8, 0x34, 0x41, 0x28,
    0xe0, 0x60, 0x12, 0x8e, 0xde, 0x35, 0x23, 0xa2,
    0x4a, 0x46, 0x1c, 0x89, 0x43, 0xab, 0x08, 0x59,
]

```

Credentials

Credentials have one possible types: `SECP256K1Credential`. Each credential is paired with an Input or Operation. The order of the credentials match the order of the inputs or operations.

SECP256K1 Credential

A `secp256k1` credential contains a list of 65-byte recoverable signatures.

What SECP256K1 Credential Contains

- `TypeID` is the ID for this type. It is `0x00000009`.
- `Signatures` is an array of 65-byte recoverable signatures. The order of the signatures must match the input's signature indices.

Gantt SECP256K1 Credential Specification

```
+-----+-----+
| type_id      : int          |        4 bytes |
+-----+-----+
| signatures    : [] [65]byte | 4 + 65 * len(signatures) bytes |
+-----+-----+
|           | 8 + 65 * len(signatures) bytes |
+-----+
```

Proto SECP256K1 Credential Specification

```
message SECP256K1Credential {
    uint32 TypeID = 1;           // 4 bytes
    repeated bytes signatures = 2; // 4 bytes + 65 bytes * len(signatures)
}
```

SECP256K1 Credential Example

Let's make a payment input with:

- **signatures :**
- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f303132333435363738393a3b3c3d3e
- 0x404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5e5d5f606162636465666768696a6b6c6e6d6f707172737475767778797a7b7c7d7e

```
[
  Signatures <- [
    0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f303132333435363738393a3b3c3d3e3f(
      0x404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5e5d5f606162636465666768696a6b6c6e6d6f707172737475767778797a7b7c7d7e7f(
    )
  ]
  =
  [
    // Type ID
    0x00, 0x00, 0x00, 0x09,
    // length:
    0x00, 0x00, 0x00, 0x02,
    // sig[0]
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1e, 0x1d, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2e, 0x2d, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
    0x00,
    // sig[1]
    0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e, 0x4f,
    0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
    0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5e, 0x5d, 0x5f,
    0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
    0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6e, 0x6d, 0x6f,
    0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
    0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e, 0x7f,
    0x00,
  ]
]
```

Signed Transaction

A signed transaction is an unsigned transaction with the addition of an array of credentials.

What Signed Transaction Contains

A signed transaction contains a `CodecID`, `UnsignedTx`, and `Credentials`.

- **CodecID** The only current valid codec id is `00 00`.
- **UnsignedTx** is an unsigned transaction, as described above.
- **Credentials** is an array of credentials. Each credential will be paired with the input in the same index at this credential.

Gantt Signed Transaction Specification

```

+-----+
| codec_id      : uint16          |           2 bytes |
+-----+
| unsigned_tx   : UnsignedTx     |           size(unsigned_tx) bytes |
+-----+
| credentials   : []Credential  |           4 + size(credentials) bytes |
+-----+
|               |           6 + size(unsigned_tx) + len(credentials) bytes |
+-----+

```

Proto Signed Transaction Specification

```

message Tx {
    uint32 codec_id = 1;           // 2 bytes
    UnsignedTx unsigned_tx = 2;    // size(unsigned_tx)
    repeated Credential credentials = 3; // 4 bytes + size(credentials)
}

```

Signed Transaction Example

Let's make a signed transaction that uses the unsigned transaction and credential from the previous examples.

- **CodecID** : 0
- **UnsignedTx** :

```
0x0000000100000003ffffffffeeeeeeeaddddcccccbbbbbbaaaaaaaaa9999999888888800000001000102030405060708090a0b0c0d0e0f10111/
```
- **Credentials**

```
0x000000090000002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f3031323334353637
```

```

[
    CodecID      <- 0x0000
    UnsignedTx   <-
0x0000000100000003ffffffffeeeeeeeaddddcccccbbbbbbaaaaaaaaa9999999888888800000001000102030405060708090a0b0c0d0e0f101112131
    Credentials <- [
        0x000000090000002000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1e1d1f202122232425262728292a2b2c2e2d2f3031323334353637
    ]
]
=
[
    // Codec ID
    0x00, 0x00,
    // unsigned transaction:
    0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x03,
    0xff, 0xff, 0xff, 0xff, 0xee, 0xee, 0xee,
    0xdd, 0xdd, 0xdd, 0xcc, 0xcc, 0xcc, 0xcc,
    0xbb, 0xbb, 0xbb, 0xaa, 0xaa, 0xaa, 0xaa,
    0x99, 0x99, 0x99, 0x88, 0x88, 0x88, 0x88,
    0x00, 0x00, 0x00, 0x01, 0x00, 0x01, 0x02, 0x03,
    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
    0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13,
    0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
    0x1c, 0x1d, 0x1e, 0x1f, 0x00, 0x00, 0x07,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x39,
    0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
    0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x02,
    0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,
    0xf6, 0x93, 0x34, 0x8f, 0x34, 0xbe, 0x6d, 0xd2,
    0x6d, 0x55, 0xa9, 0x55, 0xc3, 0x34, 0x41, 0x28,
    0xe0, 0x60, 0x12, 0x8e, 0xde, 0x35, 0x23, 0xa2,
    0x4a, 0x46, 0x1c, 0x89, 0x43, 0xab, 0x08, 0x59,
    0x00, 0x00, 0x00, 0x01, 0xf1, 0xe1, 0xd1, 0xc1,
    0xb1, 0xa1, 0x91, 0x81, 0x71, 0x61, 0x51, 0x41,
    0x31, 0x21, 0x11, 0x01, 0xf0, 0xe0, 0xd0, 0xc0,
    0xb0, 0xa0, 0x90, 0x80, 0x70, 0x60, 0x50, 0x40,
    0x30, 0x20, 0x10, 0x00, 0x00, 0x00, 0x00, 0x05,
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
    0x07, 0x5b, 0xcd, 0x15, 0x00, 0x00, 0x00, 0x02,
    0x00, 0x00, 0x00, 0x04, 0x00, 0x01, 0x02, 0x03
    // number of credentials:
    0x00, 0x00, 0x00, 0x01,
]
```

```
// credential[0]:
0x00, 0x00, 0x00, 0x09, 0x00, 0x00, 0x00, 0x02,
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1e, 0x1d, 0x1f,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2e, 0x2d, 0x2f,
0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
0x00, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46,
0x47, 0x48, 0x49, 0x4a, 0x4b, 0x4c, 0x4d, 0x4e,
0x4f, 0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56,
0x57, 0x58, 0x59, 0x5a, 0x5b, 0x5c, 0x5e, 0x5d,
0x5f, 0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66,
0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x6e, 0x6d,
0x6f, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76,
0x77, 0x78, 0x79, 0x7a, 0x7b, 0x7c, 0x7d, 0x7e,
0x7f, 0x00,
```

UTXO

A UTXO is a standalone representation of a transaction output.

What UTXO Contains

A UTXO contains a `CodecID`, `TxID`, `UTXOIndex`, and `Output`.

- `CodecID` The only current valid codec id is `00 00`.
- `TxID` is a 32-byte transaction ID. Transaction IDs are calculated by taking sha256 of the bytes of the signed transaction.
- `UTXOIndex` is an int that specifies which output in the transaction specified by `TxID` that this utxo was created by.
- `AssetID` is a 32-byte array that defines which asset this utxo references.
- `Output` is the output object that created this utxo. The serialization of Outputs was defined above.

Gantt UTXO Specification

codec_id	: uint16	2 bytes
tx_id	: [32]byte	32 bytes
output_index	: int	4 bytes
asset_id	: [32]byte	32 bytes
output	: Output	size(output) bytes
		70 + size(output) bytes

Proto UTXO Specification

```
message Utxo {
    uint32 codec_id = 1;      // 02 bytes
    bytes tx_id = 2;          // 32 bytes
    uint32 output_index = 3; // 04 bytes
    bytes asset_id = 4;       // 32 bytes
    Output output = 5;        // size(output)
}
```

UTXO Example

Let's make a UTXO from the signed transaction created above:

- `CodecID` : 0
- `TxID` : `0xf966750f438867c3c9828ddcdbe660e21ccdbb36a9276958f011ba472f75d4e7`
- `UTXOIndex` : `0x00000000`
- `AssetID` : `0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1elf`
- `Output` : "Example SECP256K1 Transferable Output as defined above"

```
[  
  CodecID   <- 0x0000  
  TxID      <- 0xf966750f438867c3c9828ddcdbe660e21ccdbb36a9276958f011ba472f75d4e7  
  UTXOIndex <- 0x00000000  
  AssetID   <- 0x000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1elf
```

```

Output      <-
0x0000000700000000000000003039000000000000d4310000001000000251025c61fbfcf078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c89c
]
=
[
// Codec ID:
0x00, 0x00,
// txID:
0xf9, 0x66, 0x75, 0x0f, 0x43, 0x88, 0x67, 0xc3,
0xc9, 0x82, 0x8d, 0xdc, 0xdb, 0xe6, 0x60, 0xe2,
0x1c, 0xcd, 0xcb, 0x36, 0xa9, 0x27, 0x69, 0x58,
0xf0, 0x11, 0xba, 0x47, 0x2f, 0x75, 0xd4, 0xe7,
// utxo index:
0x00, 0x00, 0x00, 0x00,
// assetID:
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
// output:
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x00, 0x02, 0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
0x0c, 0x0d, 0x0e, 0xf, 0x10, 0x11, 0x12, 0x13,
0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b,
0x1c, 0x1d, 0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23,
0x24, 0x25, 0x26, 0x27,
]

```

StakeableLockIn

A StakeableLockIn is a staked and locked input. The StakeableLockIn can only fund StakeableLockOuts with the same address until its lock time has passed.

What StakeableLockIn Contains

A StakeableLockIn contains a `TypeID`, `Locktime` and `TransferableIn`.

- `TypeID` is the ID for this output type. It is `0x00000015`.
- `Locktime` is a long that contains the Unix timestamp before which the input can be consumed only to stake. The Unix timestamp is specific to the second.
- `TransferableIn` is a transferable input object.

Gantt StakeableLockIn Specification

+-----+	+-----+	+-----+
type_id : int		4 bytes
+-----+	+-----+	+-----+
locktime : long		8 bytes
+-----+	+-----+	+-----+
transferable_in : TransferableInput	size(transferable_in)	
+-----+	+-----+	+-----+
	12 + size(transferable_in) bytes	
+-----+	+-----+	+-----+

Proto StakeableLockIn Specification

```

message StakeableLockIn {
    uint32 type_id = 1;           // 04 bytes
    uint64 locktime = 2;          // 08 bytes
    TransferableInput transferable_in = 3; // size(transferable_in)
}

```

StakeableLockIn Example

Let's make a StakeableLockIn with:

- `TypeID` : 21
- `Locktime` : 54321
- `TransferableIn` : "Example SECP256K1 Transfer Input as defined above"

```
[
    TypeID    <- 0x00000015
    Locktime  <- 0x000000000000d431
]
```

```

TransferableIn <- [
  f1e1d1c1b1a191817161514131211101f0e0d0c0b0a0908070605040302010000000005000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c
]
]
=
[
  // type_id:
  0x00, 0x00, 0x00, 0x15,
  // locktime:
  0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
  // transferable_in
  0xf1, 0xe1, 0xd1, 0xc1, 0xb1, 0xa1, 0x91, 0x81,
  0x71, 0x61, 0x51, 0x41, 0x31, 0x21, 0x11, 0x01,
  0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
  0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00,
  0x00, 0x00, 0x00, 0x05,
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
  0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
  0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
  0x00, 0x00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00,
  0x07, 0x5b, 0xcd, 0x15, 0x00, 0x00, 0x01,
  0x00, 0x00, 0x00, 0x00,
]
]

```

StakeableLockOut

A StakeableLockOut is an output that is locked until its lock time, but can be staked in the meantime.

What StakeableLockOut Contains

A StakeableLockOut contains a `TypeID`, `Locktime` and `TransferableOut`.

- `TypeID` is the ID for this output type. It is `0x00000016`.
- `Locktime` is a long that contains the Unix timestamp before which the output can be consumed only to stake. The Unix timestamp is specific to the second.
- `transferableout` : "Example SECP256K1 Transfer Output as defined above"

Gantt StakeableLockOut Specification

<code> type_id : int </code>	<code>4 bytes </code>
<code>+-----+-----+</code>	<code>+-----+</code>
<code> locktime : long </code>	<code>8 bytes </code>
<code>+-----+-----+</code>	<code>+-----+</code>
<code> transferable_out : TransferableOutput </code>	<code>size(transferable_out) </code>
<code>+-----+-----+</code>	<code>+-----+</code>
	<code> 12 + size(transferable_out) bytes </code>
<code>+-----+-----+</code>	<code>+-----+</code>

Proto StakeableLockOut Specification

```

message StakeableLockOut {
  uint32 type_id = 1;           // 04 bytes
  uint64 locktime = 2;          // 08 bytes
  TransferableOutput transferable_out = 3; // size(transferable_out)
}

```

StakeableLockOut Example

Let's make a stakeablelockout with:

- `TypeID` : 22
- `Locktime` : 54321
- `TransferableOutput` : "Example SECP256K1 Transfer Output from above"

```

[
  TypeID      <- 0x00000016
  Locktime    <- 0x000000000000d431
  TransferableOutput <-
  0x000000070000000000003039000000000000d4310000001000000251025c61fbfcfc078f69334f834be6dd26d55a955c3344128e060128ede3523a24a461c894
]
=
[

```

```

// type_id:
0x00, 0x00, 0x00, 0x16,
// locktime:
0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,
// transferable_out
0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00,
0x00, 0x00, 0x30, 0x39, 0x00, 0x00, 0x00,
0x00, 0x00, 0xd4, 0x31, 0x00, 0x00, 0x00, 0x01,
0x00, 0x00, 0x02, 0x51, 0x02, 0x5c, 0x61,
0xfb, 0xcf, 0xc0, 0x78, 0xf6, 0x93, 0x34, 0xf8,
0x34, 0xbe, 0x6d, 0xd2, 0x6d, 0x55, 0xa9, 0x5f,
0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,
0xde, 0x35, 0x23, 0xa2, 0xa4, 0x46, 0x1c, 0x89,
0x43, 0xab, 0x08, 0x59,
]

```

Subnet Auth

What Subnet Auth Contains

Specifies the addresses whose signatures will be provided to demonstrate that the owners of a Subnet approve something.

- **TypeID** is the ID for this type. It is `0x0000000a`.
- **AddressIndices** defines which addresses' signatures will be attached to this transaction. `AddressIndices[i]` is the index in a Subnet owner list that corresponds to the signature at index `i` in the signature list. Must be sorted low to high and not have duplicates.

Gantt Subnet Auth Specification

<code> type_id : int</code>	<code> 4 bytes </code>
<code>+-----+-----+</code>	<code>+-----+-----+</code>
<code> address_indices : []int</code>	<code> 4 + 4*len(address_indices) bytes </code>
<code>+-----+-----+</code>	<code>+-----+-----+</code>
<code> </code>	<code>8 + 4*len(address_indices) bytes </code>
<code>+-----+-----+</code>	<code>+-----+-----+</code>

Proto Subnet Auth Specification

```

message SubnetAuth {
    uint32 type_id = 1;           // 04 bytes
    repeated AddressIndex address_indices = 2; // 04 + 4*len(address_indices) bytes
}

```

Subnet Auth Example

Let's make a Subnet auth:

- **TypeID** : 10
- **AddressIndices** : [0]

```

[
    TypeID          <- 0x0000000a
    AddressIndices <- [
        0x00000000
    ]
]

=
[
    // type id
    0x00, 0x00, 0x00, 0xa,
    // num address indices
    0x00, 0x00, 0x00, 0x01,
    // address index 1
    0x00, 0x00, 0x00, 0x00
]

```

Validator

A validator verifies transactions on a blockchain.

What Validator Contains

A validator contains `NodeID`, `Start`, `End`, and `Wght`

- `NodeID` is the ID of the validator
- `Start` Unix time this validator starts validating
- `End` Unix time this validator stops validating
- `Wght` Weight of this validator used when sampling

Gantt Validator Specification

```
+-----+-----+
| node_id : string | 20 bytes |
+-----+-----+
| start   : uint64 | 8 bytes  |
+-----+-----+
| end     : uint64 | 8 bytes  |
+-----+-----+
| wght    : uint64 | 8 bytes  |
+-----+-----+
|                   | 44 bytes |
+-----+-----+
```

Proto Validator Specification

```
message Validator {
    string node_id = 1;           // 20 bytes
    uint64 start = 2;            // 8 bytes
    uint64 end = 3;              // 8 bytes
    uint64 wght = 4;             // 8 bytes
}
```

Validator Example

Let's make a validator:

- `NodeID` : "NodeID-GWPcbFJZFFZreETSoWjPimr846mXEKCTu"
- `Start` : 1643068824
- `End` : 1644364767
- `Wght` : 20

```
[
    NodeID  <- 0xa18d3991cf637aa6c162f5e95cf163f69cd8291
    Start    <- 0x61ef3d98
    End      <- 0x620303df
    Wght     <- 0x14
]

=
[
    // node id
    0xaa, 0x18, 0xd3, 0x99, 0x1c, 0xf6, 0x37,
    0xaa, 0x6c, 0x16, 0x2f, 0x5e, 0x95, 0xcf,
    0x16, 0x3f, 0x69, 0xcd, 0x82, 0x91,
    // start
    0x61, 0xef, 0x3d, 0x98,
    // end
    0x62, 0x03, 0x03, 0xdf,
    // wght
    0x14,
]
```

Rewards Owner

Where to send staking rewards when done validating

What Rewards Owner Contains

A rewards owner contains a `TypeID`, `Locktime`, `Threshold`, and `Addresses`.

- `TypeID` is the ID for this validator. It is `0x0000000b`.
- `Locktime` is a long that contains the Unix timestamp that this output can be spent after. The Unix timestamp is specific to the second.
- `Threshold` is an int that names the number of unique signatures required to spend the output. Must be less than or equal to the length of `Addresses`. If `Addresses` is empty, must be 0.
- `Addresses` is a list of unique addresses that correspond to the private keys that can be used to spend this output. Addresses must be sorted lexicographically.

Gantt Rewards Owner Specification

```
+-----+-----+
| type_id : int          | 4 bytes           |
+-----+-----+
| locktime : long         | 8 bytes           |
+-----+-----+
| threshold : int         | 4 bytes           |
+-----+-----+
| addresses : [][20]byte | 4 + 20 * len(addresses) bytes |
+-----+-----+
|                           | 40 bytes           |
+-----+-----+
```

Proto Rewards Owner Specification

```
message RewardsOwner {
    string type_id = 1;           // 4 bytes
    uint64 locktime = 2;          // 8 bytes
    uint32 threshold = 3;         // 4 bytes
    repeated bytes addresses = 4; // 4 bytes + 20 bytes * len(addresses)
}
```

Rewards Owner Example

Let's make a rewards owner:

- **TypeID** : 11
- **Locktime** : 54321
- **Threshold** : 1
- **Addresses** :
- 0x51025c61fbfcfc078f69334f834be6dd26d55a955
- 0xc3344128e060128ede3523a24a461c8943ab0859

```
[  
    TypeID <- 0x0000000b  
    Locktime <- 0x000000000000d431  
    Threshold <- 0x00000001  
    Addresses <- [  
        0x51025c61fbfcfc078f69334f834be6dd26d55a955,  
        0xc3344128e060128ede3523a24a461c8943ab0859,  
    ]  
]  
  
=  
[  
    // type id  
    0x00, 0x00, 0x00, 0x0b,  
    // locktime  
    0x00, 0x00, 0x00, 0x00, 0x00, 0xd4, 0x31,  
    // threshold:  
    0x00, 0x00, 0x00, 0x01,  
    // number of addresses:  
    0x00, 0x00, 0x00, 0x02,  
    // addrs[0]:  
    0x51, 0x02, 0x5c, 0x61, 0xfb, 0xcf, 0xc0, 0x78,  
    0xf6, 0x93, 0x34, 0xf8, 0x34, 0xbe, 0x6d, 0xd2,  
    0x6d, 0x55, 0xa9, 0x55,  
    // addrs[1]:  
    0xc3, 0x34, 0x41, 0x28, 0xe0, 0x60, 0x12, 0x8e,  
    0xde, 0x35, 0x23, 0xa2, 0x4a, 0x46, 0x1c, 0x89,  
    0x43, 0xab, 0x08, 0x59,  
]
```

Serialization Primitives

Avalanche uses a simple, uniform, and elegant representation for all internal data. This document describes how primitive types are encoded on the Avalanche platform. Transactions are encoded in terms of these basic primitive types.

Byte

Bytes are packed as-is into the message payload.

Example:

```
Packing:  
    0x01  
Results in:  
    [0x01]
```

Short

Shorts are packed in BigEndian format into the message payload.

Example:

```
Packing:  
    0x0102  
Results in:  
    [0x01, 0x02]
```

Integer

Integers are 32-bit values packed in BigEndian format into the message payload.

Example:

```
Packing:  
    0x01020304  
Results in:  
    [0x01, 0x02, 0x03, 0x04]
```

Long Integers

Long integers are 64-bit values packed in BigEndian format into the message payload.

Example:

```
Packing:  
    0x0102030405060708  
Results in:  
    [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]
```

IP Addresses

IP addresses are represented as 16-byte IPv6 format, with the port appended into the message payload as a Short. IPv4 addresses are padded with 12 bytes of leading 0x00s.

IPv4 example:

```
Packing:  
    "127.0.0.1:9650"  
Results in:  
    [  
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
        0x00, 0x00, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x01,  
        0x25, 0xb2,  
    ]
```

IPv6 example:

```
Packing:  
    "[2001:0db8:ac10:fe01::]:12345"  
Results in:  
    [  
        0x20, 0x01, 0x0d, 0xb8, 0xac, 0x10, 0xfe, 0x01,  
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
        0x30, 0x39,  
    ]
```

Fixed-Length Array

Fixed-length arrays, whose length is known ahead of time and by context, are packed in order.

Byte array example:

```
Packing:  
    [0x01, 0x02]
```

```
Results in:  
[0x01, 0x02]
```

Integer array example:

```
Packing:  
[0x03040506]  
Results in:  
[0x03, 0x04, 0x05, 0x06]
```

Variable Length Array

The length of the array is prefixed in Integer format, followed by the packing of the array contents in Fixed Length Array format.

Byte array example:

```
Packing:  
[0x01, 0x02]  
Results in:  
[0x00, 0x00, 0x00, 0x02, 0x01, 0x02]
```

Int array example:

```
Packing:  
[0x03040506]  
Results in:  
[0x00, 0x00, 0x00, 0x01, 0x03, 0x04, 0x05, 0x06]
```

String

A String is packed similarly to a variable-length byte array. However, the length prefix is a short rather than an int. Strings are encoded in UTF-8 format.

Example:

```
Packing:  
"Avax"  
Results in:  
[0x00, 0x04, 0x41, 0x76, 0x61, 0x78]
```

Avalanche Subnet Faucet

Right now there are thousands of networks and chains in the blockchain space, each with its capabilities and use-cases. And each network requires native coins to do any transaction on them, which can have a monetary value as well. These coins can be collected through centralized exchanges, token sales, etc in exchange for some monetary assets like USD.

But we cannot risk our funds on the network or on any applications hosted on that network, without testing them first. So, these networks often have test networks or testnets, where the native coins do not have any monetary value, and thus can be obtained freely through faucets.

These testnets are often the testbeds for any new native feature of the network itself, or any dapp or [Subnet](#) that is going live on the main network (Mainnet). For example, [Fuji](#) network is the Testnet for Avalanche's Mainnet.

Besides Fuji Testnet, [Avalanche Faucet](#) can be used to get free coins on these testnets Subnets like -

- [WAGMI Testnet](#)
- [DeFi Kingdoms Testnet](#)
- [Dexalot Testnet](#) and many more.

You can use this [repository](#) to deploy your faucet or just make a PR with the [configurations](#) of the Subnet. This faucet comes with many features like multiple chain support, custom rate-limiting per Subnet, CAPTCHA verification, and concurrent transaction handling.

Summary

A [Faucet](#) powered by Avalanche for Fuji Network and other Subnets. You can -

- Request test coins for the supported Subnets
- Integrate your EVM Subnet with the faucet by making a PR with the [chain configurations](#)
- Fork the [repository](#) to deploy your faucet for any EVM chain

Adding a New Subnet

You can also integrate a new Subnet on the live [faucet](#) with just a few lines of configuration parameters. All you have to do is make a PR on the [Avalanche Faucet](#) git repository with the Subnet's information. The following parameters are required.

```
{  
  "ID": string,  
  "NAME": string,
```

```

    "TOKEN": string,
    "RPC": string,
    "CHAINID": number,
    "EXPLORER": string,
    "IMAGE": string,
    "MAX_PRIORITY_FEE": string,
    "MAX_FEE": string,
    "DRIP_AMOUNT": number,
    "RATELIMIT": {
      "MAX_LIMIT": number,
      "WINDOW_SIZE": number
    }
}

```

- **ID** - Each Subnet chain should have a unique and relatable ID.
- **NAME** - Name of the Subnet chain that will appear on the site.
- **RPC** - A valid RPC URL for accessing the chain.
- **CHAINID** - ChainID of the chain
- **EXPLORER** - Base URL of standard explorer's site.
- **IMAGE** - URL of the icon of the chain that will be shown in the dropdown.
- **MAX_PRIORITY_FEE** - Maximum tip per faucet drop in **wei** or **10^{-18}** unit (for EIP1559 supported chains)
- **MAX_FEE** - Maximum fee that can be paid for a faucet drop in **wei** or **10^{-18}** unit
- **DRIP_AMOUNT** - Amount of coins to send per request in **gwei** or **10^{-9}** unit
- **RECALIBRATE** (*optional*) - Number of seconds after which the nonce and balance will recalibrate
- **RATELIMIT** - Number of times (MAX_LIMIT) to allow per user within the WINDOW_SIZE (in minutes)

Add the configuration in the array of `evmchains` inside the [config.json](#) file and make a PR.

Building and Deploying a Faucet

You can also deploy and build your faucet by using the [Avalanche Faucet](#) repository.

Requirements

- [Node](#) >= 17.0 and [npm](#) >= 8.0
- [Google's reCAPTCHA](#) v3 keys
- [Docker](#)

Installation

Clone this repository at your preferred location.

```
git clone https://github.com/ava-labs/avalanche-faucet
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanche-faucet.git
```

You can find more about SSH and how to use it [here](#). :::

Client-Side Configurations

We need to configure our application with the server API endpoints and CAPTCHA site keys. All the client-side configurations are there in the `client/src/config.json` file. Since there are no secrets on the client-side, we do not need any environment variables. Update the config files according to your need.

```
{
  "banner": "/banner.png",
  "apiBaseEndpointProduction": "/api/",
  "apiBaseEndpointDevelopment": "http://localhost:8000/api/",
  "apiTimeout": 10000,
  "CAPTCHA": {
    "siteKey": "6LcNSCYfAAAAAJH8fauA-okTZrmAxYqfF9gOmujf",
    "action": "faucetdrip"
  }
}
```

Put the Google's reCAPTCHA site-key without which the faucet client can't send the necessary CAPTCHA response to the server. This key is not a secret and could be public.

In the above file, there are 2 base endpoints for the faucet server `apiBaseEndpointProduction` and `apiBaseEndpointDevelopment`.

In production mode, the client-side will be served as static content over the server's endpoint, and hence we do not have to provide the server's IP address or domain.

The URL path should be valid, where the server's APIs are hosted. If the endpoints for API have a leading `/v1/api` and the server is running on localhost at port 3000, then you should use `http://localhost:3000/v1/api` or `/v1/api/` depending on whether it is production or development.

Server-Side Configurations

On the server-side, we need to configure 2 files - `.env` for secret keys and `config.json` for chain and API rate limiting configurations.

Setup Environment Variables

Setup the environment variable with your private key and reCAPTCHA secret. Make a `.env` file in your preferred location with the following credentials, as this file will not be committed to the repository. The faucet server can handle multiple EVM chains, and therefore requires private keys for addresses with funds on each of the chains.

If you have funds on the same address on every chain, then you can specify them with the single variable `PK`. But if you have funds on different addresses on different chains, then you can provide each of the private keys against the ID of the chain, as shown below.

```
C="C chain private key"
WAGMI="Wagmi chain private key"
PK="Sender Private Key with Funds in it"
CAPTCHA_SECRET="Google reCAPTCHA Secret"
```

`PK` will act as a fallback private key, in case, the key for any chain is not provided.

Setup EVM Chain Configurations

You can create a faucet server for any EVM chain by making changes in the `config.json` file. Add your chain configuration as shown below in the `evmchains` object. Configuration for Fuji's C-Chain and WAGMI chain is shown below for example.

```
"evmchains": [
  {
    "ID": "C",
    "NAME": "Fuji (C-Chain)",
    "TOKEN": "AVAX",
    "RPC": "https://api.avax-test.network/ext/C/rpc",
    "CHAINID": 43113,
    "EXPLORER": "https://testnet.snowtrace.io",
    "IMAGE": "/avaxred.png",
    "MAX_PRIORITY_FEE": "2000000000",
    "MAX_FEE": "100000000000",
    "DRIP_AMOUNT": 2000000000,
    "RECALIBRATE": 30,
    "RATELIMIT": {
      "MAX_LIMIT": 1,
      "WINDOW_SIZE": 1440
    }
  },
  {
    "ID": "WAGMI",
    "NAME": "WAGMI Testnet",
    "TOKEN": "WGM",
    "RPC": "https://subnets.avax.network/wagmi/wagmi-chain-testnet/rpc",
    "CHAINID": 11111,
    "EXPLORER": "https://subnets.avax.network/wagmi/wagmi-chain-testnet/explorer",
    "IMAGE": "/wagmi.png",
    "MAX_PRIORITY_FEE": "2000000000",
    "MAX_FEE": "100000000000",
    "DRIP_AMOUNT": 2000000000,
    "RATELIMIT": {
      "MAX_LIMIT": 1,
      "WINDOW_SIZE": 1440
    }
  }
]
```

In the above configuration drip amount is in `nAVAX` or `gwei`, whereas fees are in `wei`. For example, with the above configurations, the faucet will send `1 AVAX` with maximum fees per gas being `100 nAVAX` and priority fee as `2 nAVAX`.

The rate limiter for C-Chain will only accept 1 request in 60 minutes for a particular API and 2 requests in 60 minutes for the WAGMI chain. Though it will skip any failed requests so that users can request tokens again, even if there is some internal error in the application. On the other hand, the global rate limiter will allow 15 requests per minute on every API. This time failed requests will also get counted so that no one can abuse the APIs.

API Endpoints

This server will expose the following APIs

Health API

The `/health` API will always return a response with a `200` status code. This endpoint can be used to know the health of the server.

```
curl http://localhost:8000/health
```

Response

```
Server healthy
```

Get Faucet Address

This API will be used for fetching the faucet address.

```
curl http://localhost:8000/api/faucetAddress?chain=C
```

It will give the following response

```
0x3EA53fA26b41885cB9149B62f0b7c0BAf76C78D4
```

Get Faucet Balance

This API will be used for fetching the faucet address.

```
curl http://localhost:8000/api/getBalance?chain=C
```

It will give the following response

```
14282900936
```

Send Token

This API endpoint will handle token requests from users. It will return the transaction hash as a receipt of the faucet drip.

```
curl -d '{  
    "address": "0x3EA53fA26b41885cB9149B62f0b7c0BAf76C78D4"  
    "chain": "C"  
}' -H 'Content-Type: application/json' http://localhost:8000/api/sendToken
```

Send token API requires a CAPTCHA response token that is generated using the CAPTCHA site key on the client-side. Since we can't generate and pass this token while making a curl request, we have to disable the CAPTCHA verification for testing purposes. You can find the steps to disable it in the next sections. The response is shown below

```
{  
    "message": "Transaction successful on Avalanche C Chain!",  
    "txHash": "0x3d1f1c3facf59c5cd7d6937b3b727d047a1e664f52834daf20b0555e89fc8317"  
}
```

Rate Limiters (Important)

The rate limiters are applied on the global (all endpoints) as well as on the `/api/sendToken` API. These can be configured from the `config.json` file. Rate limiting parameters for chains are passed in the chain configuration as shown above.

```
"GLOBAL_RL": {  
    "ID": "GLOBAL",  
    "RATELIMIT": {  
        "REVERSE_PROXY": 4,  
        "MAX_LIMIT": 40,  
        "WINDOW_SIZE": 1,  
        "PATH": "/",
        "SKIP_FAILED_REQUESTS": false  
    }  
}
```

There could be multiple proxies between the server and the client. The server will see the IP address of the adjacent proxy connected with the server, and this may not be the client's actual IP.

The IPs of all the proxies that the request has hopped through are stuffed inside the header `x-forwarded-for` array. But the proxies in between can easily manipulate these headers to bypass rate limiters. So, we cannot trust all the proxies and hence all the IPs inside the header.

The proxies that are set up by the owner of the server (reverse-proxies) are the trusted proxies on which we can rely and know that they have stuffed the actual IP of the callers in between. Any proxy that is not set up by the server, should be considered an untrusted proxy. So, we can jump to the IP address added by the last proxy that we trust. The number of jumps that we want can be configured in the `config.json` file inside the `GLOBAL_RL` object.



Clients Behind Same Proxy

Consider the below diagram. The server is set up with 2 reverse proxies. If the client is behind proxies, then we cannot get the client's actual IP, and instead will consider the proxy's IP as the client's IP. And if some other client is behind the same proxy, then those clients will be considered as a single entity and might get

rate-limited faster.



Therefore it is advised to the users, to avoid using any proxy for accessing applications that have critical rate limits, like this faucet.

Wrong Number of Reverse Proxies

So, if you want to deploy this faucet, and have some reverse proxies in between, then you should configure this inside the `GLOBAL_RL` key of the `config.json` file. If this is not configured properly, then the users might get rate-limited very frequently, since the server-side proxy's IP addresses are being viewed as the client's IP. You can verify this in the code [here](#).

```
"GLOBAL_RL": {  
  "ID": "GLOBAL",  
  "RATELIMIT": {  
    "REVERSE_PROXYIES": 4,  
    ...  
  }  
}
```



It is also quite common to have Cloudflare as the last reverse proxy or the exposed server. Cloudflare provides a header `cf-connecting-ip` which is the IP of the client that requested the faucet and hence Cloudflare. We are using this as default.

CAPTCHA Verification

CAPTCHA is required to prove the user is a human and not a bot. For this purpose, we will use [Google's reCAPTCHA](#). The server-side will require `CAPTCHA_SECRET` that should not be exposed. You can set the threshold score to pass the CAPTCHA test by the users [here](#).

You can disable these CAPTCHA verifications and rate limiters for testing the purpose, by tweaking in the `server.ts` file.

Disabling Rate Limiters

Comment or remove these 2 lines from the `server.ts` file

```
new RateLimiter(app, [GLOBAL_RL])  
new RateLimiter(app, evmchains)
```

Disabling CAPTCHA Verification

Remove the `captcha.middleware` from `sendToken` API.

Starting the Faucet

Follow the below commands to start your local faucet.

Installing Dependencies

This will concurrently install dependencies for both client and server.

```
npm install
```

If ports have a default configuration, then the client will start at port 3000 and the server will start at port 8000 while in development mode.

Starting in Development Mode

This will concurrently start the server and client in development mode.

```
npm run dev
```

Building for Production

The following command will build server and client at `build/` and `build/client` directories.

```
npm run build
```

Starting in Production Mode

This command should only be run after successfully building the client and server-side code.

```
npm start
```

Setting up with Docker

Follow the steps to run this application in a Docker container.

Build Docker Image

Docker images can be served as the built versions of our application, that can be used to deploy on Docker container.

```
docker build . -t faucet-image
```

Starting Application inside Docker Container

Now we can create any number of containers using the above `faucet` image. We also have to supply the `.env` file or the environment variables with the secret keys to create the container. Once the container is created, these variables and configurations will be persisted and can be easily started or stopped with a single command.

```
docker run -p 3000:8000 --name faucet-container --env-file ./env faucet-image
```

The server will run on port 8000, and our Docker will also expose this port for the outer world to interact. We have exposed this port in the `Dockerfile`. But we cannot directly interact with the container port, so we had to bind this container port to our host port. For the host port, we have chosen 3000. This flag `-p 3000:8000` achieves the same.

This will start our faucet application in a Docker container at port 3000 (port 8000 on the container). You can interact with the application by visiting <http://localhost:3000> in your browser.

Stopping the Container

You can easily stop the container using the following command

```
docker stop faucet-container
```

Restarting the Container

To restart the container, use the following command

```
docker start faucet-container
```

Using the Faucet

Using the faucet is quite straightforward, but for the sake of completeness, let's go through the steps, to collect your first test coins.

Visit Avalanche Faucet Site

Go to <https://faucet.avax.network>. You will see various network parameters like network name, faucet balance, drop amount, drop limit, faucet address, etc.



Select Network

You can use the dropdown to select the network of your choice and get some free coins (each network may have a different drop amount).



Put Address and Request Coins

Put your wallet address where you want to get a drop, and click the **Request** button. Within a second, you will get a **transaction hash** for the processed transaction. The hash would be a hyperlink to Subnet's explorer. You can see the transaction status, by clicking on that hyperlink.



More Interactions

This is not just it. Using the buttons shown below, you can go to the Subnet explorer or add the Subnet to your browser wallet extensions like MetaMask with a single click.



Probable Errors and Troubleshooting

Errors are not expected, but if you are facing some of the errors shown, then you could try troubleshooting as shown below. If none of the troubleshooting works, reach us through [Discord](#).

- **Too many requests. Please try again after X minutes** This is a rate-limiting message. Every Subnet can set its drop limits. The above message suggests that you have reached your drop limit, that is the number of times you could request coins within the window of X minutes. You should try requesting after X minutes. If you are facing this problem, even when you are requesting for the first time in the window, you may be behind some proxy, Wi-Fi, or VPN service that is also being used by some other user.
- **CAPTCHA verification failed! Try refreshing** We are using v3 of [Google's reCAPTCHA](#). This version uses scores between 0 and 1 to rate the interaction of humans with the site, with 0 being the most suspicious one. You do not have to solve any puzzle or mark the **I am not a Robot** checkbox. The score will be automatically calculated. We want our users to score at least 0.3 to use the faucet. This is configurable, and we will update the threshold after having

broader data. But if you are facing this issue, then you can try refreshing your page, disabling ad-blockers, or switching off any VPN. You can follow this [guide](#) to get rid of this issue.

- **Internal RPC error! Please try after sometime** This is an internal error in the Subnet's node, on which we are making an RPC for sending transactions. A regular check will update the RPC's health status every 30 seconds (default) or whatever is set in the configuration. This may happen only in rare scenarios and you cannot do much about it, rather than waiting.
- **Timeout of 10000ms exceeded** There could be many reasons for this message. It could be an internal server error, or the request didn't receive by the server, slow internet, etc. You could try again after some time, and if the problem persists, then you should raise this issue on our [Discord](#) server.
- **Couldn't see any transaction status on explorer** The transaction hash that you get for each drop is pre-computed using the expected nonce, amount, and receiver's address. Though transactions on Avalanche are near-instant, the explorer may take time to index those transactions. You should wait for a few more seconds, before raising any issue or reaching out to us.

Build Your First Subnet

The first step of learning Subnet development is learning to use [Avalanche-CLI](#).

The best way to get started is by jumping in and deploying your first Subnet.

This tutorial walks you through the process of using Avalanche-CLI for the first time by creating a Subnet, deploying it to the local network, and connecting to it with MetaMask.

Installation

The fastest way to install the latest Avalanche-CLI binary is by running the install script:

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s
```

The binary installs inside the `~/bin` directory. If the directory doesn't exist, it will be created.

You can run all of the commands in this tutorial by calling `~/bin/avalanche`.

You can also add the command to your system path by running

```
export PATH=~/bin:$PATH
```

If you add it to your path, you should be able to call the program anywhere with just `avalanche`. To add it to your path permanently, add an export command to your shell initialization script (ex: `.bashrc` or `.zshrc`).

For more detailed installation instructions, see [Avalanche-CLI Installation](#)

Create Your Subnet Configuration

This tutorial teaches you how to create an Ethereum Virtual Machine (EVM) based Subnet. To do so, you use Subnet-EVM, Avalanche's Subnet fork of the EVM. It supports airdrops, custom fee tokens, configurable gas parameters, and multiple stateful precompiles. To learn more, take a look at [Subnet-EVM](#). The goal of your first command is to create a Subnet-EVM configuration.

The Subnet command suite provides a collection of tools for developing and deploying Subnets.

The Subnet Creation Wizard walks you through the process of creating your Subnet. To get started, first pick a name for your Subnet. This tutorial uses `mySubnet`, but feel free to substitute that with any name you like. Once you've picked your name, run

```
avalanche subnet create mySubnet
```

The following sections walk through each question in the wizard.

Choose Your VM

Select `SubnetEVM`.

Enter Your Subnet's ChainID

Choose a positive integer for your EVM-style ChainID.

In production environments, this ChainID needs to be unique and not shared with any other chain. You can visit [chainlist](#) to verify that your selection is unique. Because this is a development Subnet, feel free to pick any number. Stay away from well-known ChainIDs such as 1 (Ethereum) or 43114 (Avalanche C-Chain) as those may cause issues with other tools.

Token Symbol

Enter a string to name your Subnet's native token. The token symbol doesn't necessarily need to be unique. Example token symbols are AVAX, JOE, and BTC.

Subnet-EVM Version

Select `Use latest version`.

Gas Fee Configuration

This question determines how to set gas fees on your Subnet.

Select `Low disk use / Low Throughput 1.5 mil gas/s (C-Chain's setting)`.

Airdrop

Select Airdrop 1 million tokens to the default address (do not use in production).

This address's private key is well-known, so DO NOT send any production funds to it. Attackers would likely drain the funds instantly.

When you are ready to start more mature testing, select `Customize your airdrop` to distribute funds to additional addresses.

Precompiles

Precompiles are Avalanche's way of customizing the behavior of your Subnet. They're strictly an advanced feature, so you can safely select `No` for now.

Wrapping Up

If all worked successfully, the command prints `Successfully created subnet configuration.`

You've successfully created your first Subnet configuration. Now it's time to deploy it.

Deploying Subnets Locally

To deploy your Subnet, run

```
avalanche subnet deploy mySubnet
```

Make sure to substitute the name of your Subnet if you used a different one than `mySubnet`.

Next, select `Local Network`.

This command boots a five node Avalanche network on your machine. It needs to download the latest versions of AvalancheGo and Subnet-EVM. The command may take a couple minutes to run.

Note: If you run `bash` on your shell and are running Avalanche-CLI on ARM64 on Mac, you will require Rosetta 2 to be able to deploy Subnets locally. You can download Rosetta 2 using `softwareupdate --install-rosetta`.

If all works as expected, the command output should look something like this:

```
> avalanche subnet deploy mySubnet
✓ Local Network
Deploying [mySubnet] to Local Network
Installing subnet-evm-v0.4.3...
subnet-evm-v0.4.3 installation successful
Backend controller started, pid: 93928, output at: /Users/subnet-developer/.avalanche-cli/runs/server_20221122_173138/avalanche-
cli-backend
Installing avalanchego-v1.9.3...
avalanchego-v1.9.3 installation successful
VMs ready.
Starting network...
.....
Blockchain has been deployed. Wait until network acknowledges...
.....
Network ready to use. Local network node endpoints:
+-----+-----+
| NODE | VM | URL | +-----+
+-----+-----+
| node2 | mySubnet | http://127.0.0.1:9652/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc | +-----+
+-----+-----+
| node3 | mySubnet | http://127.0.0.1:9654/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc | +-----+
+-----+-----+
| node4 | mySubnet | http://127.0.0.1:9656/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc | +-----+
+-----+-----+
| node5 | mySubnet | http://127.0.0.1:9658/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc | +-----+
+-----+-----+
| node1 | mySubnet | http://127.0.0.1:9650/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc | +-----+
+-----+-----+

Browser Extension connection details (any node URL from above works):
RPC URL: http://127.0.0.1:9650/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Funded address: 0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC with 1000000 (10^18) - private key:
56289e9c94b6912bf12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027
Network name: mySubnet
Chain ID: 54325
Currency Symbol: TUTORIAL
```

You can use the deployment details to connect to and interact with your Subnet. Now it's time to interact with it.

Interacting with Your Subnet

You can use the value provided by `Browser Extension connection details` to connect to your Subnet with MetaMask, Core, or any other wallet.

:::note

To allow API calls from other machines, use `--http-host=0.0.0.0` in the config.

...

```
Browser Extension connection details (any node URL from above works):
RPC URL: http://127.0.0.1:9650/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbf0V9DYApDJVXKXuwVFPz/rpc
Funded address: 0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC with 1000000 (10^18) - private key:
56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027
Network name: mySubnet
Chain ID: 54325
Currency Symbol: TUTORIAL
```

This tutorial uses MetaMask.

Importing the Test Private Key

:::warning This address derives from a well-known private key. Anyone can steal funds sent to this address. Only use it on development networks that only you have access to. If you send production funds to this address, attackers may steal them instantly. :::

First, you need to import your airdrop private key into MetaMask. Do this by clicking the profile bubble in the top right corner and select `Import account`.



Import the well-known private key `0x56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027`.



Next, rename the MetaMask account to prevent confusion. Switch to the new account and click the three dot menu in the top right corner. Select `Account details`. Click the edit icon next to the account's name. Rename the account `DO NOT USE -- Public test key` to prevent confusion with any personal wallets.



Click the checkmark to confirm the change.

Connect to the Subnet

Next, you need to add your Subnet to MetaMask's networks.

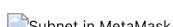
Click the profile bubble in the top right corner and select `Settings`. Next, click `Networks`. Finally, click `Add network`.



At the bottom of the next page, select `Add a network manually`. Enter your Subnet's details, found in the output of your `avalanche subnet deploy command`, into the form and click `Save`.



If all worked as expected, your balance should read 1 million tokens. Your Subnet is ready for action. You might want to try to [Deploy a Smart Contract on Your Subnet-EVM Using Remix and MetaMask](#).



Next Steps

Congrats Subnetooooor, you just deployed your first Subnet!

After you feel comfortable with this deployment flow, try deploying smart contracts on your chain with [Remix](#), [Hardhat](#), or [Foundry](#). You can also experiment with customizing your Subnet by adding precompiles or adjusting the airdrop.

Once you've developed a stable Subnet you like, see [Create an EVM Subnet on Fuji Testnet](#) to take your Subnet one step closer to production.

Good Subnetting!

FAQ

How is the Subnet ID determined upon creation?

The Subnet ID is the hash of the transaction that created the Subnet.

Case Study: WAGMI Upgrade

This case study uses [WAGMI](#) Subnet upgrade to show how a network upgrade on an EVM-based (Ethereum Virtual Machine) Subnet can be done simply, and how the resulting upgrade can be used to dynamically control fee structure on the Subnet.

Introduction

[Subnet-EVM](#) aims to provide an easy to use toolbox to customize the EVM for your blockchain. It is meant to run out of the box for many Subnets without any modification. But what happens when you want to add a new feature updating the rules of your EVM?

Instead of hard coding the timing of network upgrades in client code like most EVM chains, requiring coordinated deployments of new code, [Subnet-EVM v0.2.8](#) introduces the long awaited feature to perform network upgrades by just using a few lines of JSON in a configuration file.

WAGMI

The [WAGMI](#) ("We're All Going to Make It") Subnet Demo is a high throughput testbed for EVM optimizations. It is parameterized to run at a factor more capacity than Fuji/Mainnet C-Chain and will be used to experiment with release candidates before they make it into an official Coreth release.

Network Parameters

- NetworkID: 11111
- ChainID: 11111
- Block Gas Limit: 20,000,000 (2.5x C-Chain)
- 10s Gas Target: 100,000,000 (~6.67x C-Chain)
- Min Fee: 1 Gwei (4% of C-Chain)
- Target Block Rate: 2s (Same as C-Chain)

You can check out the [Genesis file of WAGMI Subnet](#) to see the initial configuration.

Network Upgrades: Enable/Disable Precompiles

Detailed description of how to do this can be found in [Customize a Subnet](#) tutorial. Here's a summary:

- Network Upgrade utilizes existing precompiles on the Subnet-EVM:
 - ContractDeployerAllowList, for restricting smart contract deployers
 - TransactionAllowList, for restricting who can submit transactions
 - NativeMinter, for minting native coins
 - FeeManager, for configuring dynamic fees
- Each of these precompiles can be individually enabled or disabled at a given timestamp as a network upgrade, or any of the parameters governing its behavior changed.
- These upgrades must be specified in a file named `upgrade.json` placed in the same directory where `config.json` resides: `{chain-config-dir}/{blockchainID}/upgrade.json`.

Preparation

To prepare for the WAGMI network upgrade, on August 15, 2022, we had announced on [Twitter](#) and shared on other social media such as Discord, with the following information:

With Subnet-EVM v0.2.8 It's time for a whole new Subnet Season: Network Upgrade Edition.

Like every great show, we're kicking this season off with a pilot episode: WAGMI Network Upgrade.

Stay tuned because this pilot is literally a can't miss for every WAGMI node 😊

The upgrade will activate the fee config manager, and enable smooth fee config updates in the future <https://docs.avax.network/subnets/customize-a-subnet#configuring-dynamic-fees>

This upgrade changes how blocks are processed on WAGMI, so every WAGMI node needs to upgrade to continue to validate WAGMI correctly.

In order to update your node, you need to update to Subnet-EVM v0.2.8 and follow the instructions to enable a stateful precompile on Subnet-EVM here: <https://docs.avax.network/subnets/customize-a-subnet#network-upgrades-enabledisable-precompiles>

You can find the JSON to configure the network upgrade in this gist: <https://gist.github.com/aaronbuchwald/b3af9da34678f542ce31717e7963085b>

TLDR; you will need to place the JSON file into your node's file directory within `chain-config-dir/wagmi blockchainID/upgrade.json` and restart your node.

Note: the WAGMI blockchainID is 2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeqh6tt.

Deploying upgrade.json

The content of the `upgrade.json` is:

```
{
  "precompileUpgrades": [
    {
      "feeManagerConfig": {
        "adminAddresses": ["0x6f0f6DA1852857d7789f68a28bba866671f3880D"],
        "blockTimestamp": 1660658400
      }
    }
  ]
}
```

Detailed explanation of feeManagerConfig can be found in the [precompiles documentation](#).

With the above `upgrade.json`, we intend to change the `adminAddresses` at timestamp `1660658400`:

- `0x6f0f6DA1852857d7789f68a28bba866671f3880D` is named as the new Admin of the FeeManager
- `1660658400` is the [Unix timestamp](#) for 10:00 AM EDT August 16, 2022 (future time when we made the announcement) when the new FeeManager change would take effect.

We place the `upgrade.json` file in the chain config directory, which in our case is

```
~/.avalanchego/configs/chains/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeqh6tt/ . After that, we restart the node so the upgrade file is loaded.
```

When the node restarts, AvalancheGo reads the contents of the JSON file and passes it into Subnet-EVM. We see a log of the chain configuration that includes the updated precompile upgrade. It looks like this:

```
INFO [08-15|15:09:36.772] <2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeqh6tt Chain>
github.com/ava-labs/subnet-evm/eth/backend.go:155: Initialised chain configuration
config=(ChainID: 11111 Homestead: 0 EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0
Constantinople: 0 Petersburg: 0 Istanbul: 0, Muir Glacier: 0, Subnet-EVM: 0, FeeConfig:
{`gasLimit`:20000000,`targetBlockRate`:2,`minBaseFee`:1000000000,`targetGas`:100000000,`baseFeeChangeDenominator`:48,`minBlockGasCost`:0,`maxBlockGasCost`:10000000,`blockGasCostStep`:500000}, AllowFeeRecipients: false, NetworkUpgrades: {\`subnetEVMTimestamp\`:0}, PrecompileUpgrade: {}, UpgradeConfig: {\`precompileUpgrades\`:[{\`feeManagerConfig\`:{`adminAddresses`:[`0x6f0f6da1852857d7789f68a28bba866671f3880D`]},`blockTimestamp\`:1660658400}}]}, Engine: Dummy Consensus Engine)"
```

We note that `precompileUpgrades` correctly shows the upcoming precompile upgrade. Upgrade is locked in and ready.

Activation

When the time passed 10:00 AM EDT August 16, 2022 (Unix timestamp 1660658400), the `upgrade.json` had been executed as planned and the new FeeManager admin address has been activated. From now on, we don't need to issue any new code or deploy anything on the WAGMI nodes to change the fee structure. Let's see how it works in practice!

Using Fee Manager

The owner `0x6f0f6DA1852857d7789f68a28bba866671f3880D` can now configure the fees on the Subnet as they see fit. To do that, all that's needed is access to the network, the private key for the newly set manager address and making calls on the precompiled contract.

We will use [Remix](#) online Solidity IDE and the [Core Browser Extension](#). Core comes with WAGMI network built-in. MetaMask will do as well but you will need to [add WAGMI](#) yourself.

First using Core, we open the account as the owner `0x6f0f6DA1852857d7789f68a28bba866671f3880D`.

Then we connect Core to WAGMI, Switch on the `Testnet Mode` in `Advanced` page in the hamburger menu:



And then open the `Manage Networks` menu in the networks dropdown. Select WAGMI there by clicking the star icon:



We then switch to WAGMI in the networks dropdown. We are ready to move on to Remix now, so we open it in the browser. First, we check that Remix sees the extension and correctly talks to it. We select `Deploy & run transactions` icon on the left edge, and on the Environment dropdown, select `Injected Provider`. We need to approve the Remix network access in the Core browser extension. When that is done, `Custom (11111) network` is shown:



Good, we're talking to WAGMI Subnet. Next we need to load the contracts into Remix. Using 'load from GitHub' option from the Remix home screen we load two contracts:

- [IAllowList.sol](#)
- and [IFeeManager.sol](#).

`IFeeManager` is our precompile, but it references the `IAllowList`, so we need that one as well. We compile `IFeeManager.sol` and deploy at the precompile address `0x020003` used on the [Subnet](#).

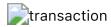


Now we can interact with the `FeeManager` precompile from within Remix via Core. For example, we can use the `getFeeConfig` method to check the current fee configuration. This action can be performed by anyone as it is just a read operation.

Once we have the new desired configuration for the fees on the Subnet, we can use the `setFeeConfig` to change the parameters. This action can **only** be performed by the owner `0x6f0f6DA1852857d7789f68a28bba866671f3880D` as the `adminAddress` specified in the [upgrade.json above](#).



When we call that method by pressing the `transact` button, a new transaction is posted to the Subnet, and we can see it on [the explorer](#):



Immediately after the transaction is accepted, the new fee config takes effect. We can check with the `getFeeConfig` that the values are reflected in the active fee config (again this action can be performed by anyone):



That's it, fees changed! No network upgrades, no complex and risky deployments, just making a simple contract call and the new fee configuration is in place!

Conclusion

Network upgrades can be complex and perilous procedures to carry out safely. Our continuing efforts with Subnets is to make upgrades as painless and simple as possible. With the powerful combination of stateful precompiles and network upgrades via the upgrade configuration files we have managed to greatly simplify both the network upgrades and network parameter changes. This in turn enables much safer experimentation and many new use cases that were too risky and complex to carry out with high-coordination efforts required with the traditional network upgrade mechanisms.

We hope this case study will help spark ideas for new things you may try on your own. We're looking forward to seeing what you have built and how easy upgrades help you in managing your Subnets! If you have any questions or issues, feel free to contact us on our [Discord](#). Or just reach out to tell us what exciting new things you have built!

Deploying Subnet with EVM Based Blockchain using AvalancheJS

Introduction

A blockchain is an instance of **Virtual Machine (VM)** that processes transactions to change the genesis (base) and the subsequent states. VMs define blockchain's state, state transition function, transactions, and the API through which users can interact with the blockchain. On Avalanche, VM serves as the blueprint for creating different instances of independent blockchains, yet sharing the same rule set.

All transactions on a blockchain need to be validated by the validators. **Subnets** are the group of validators that can validate one or many chains. On Avalanche, the primary Subnet, that currently validates 3 blockchains, and each chain serves a different purpose. One can easily deploy their Subnet and invite other validators to be a part of it. A validator can be a part of multiple Subnets, but it must be validating the primary Subnet.

On Avalanche, out of the 3 chains, **Platform Chain (P-Chain)** manages Subnets, Subnet validators, Subnet blockchain, etc. In this tutorial, you will learn about creating your Subnet and deploying **Ethereum Virtual Machine (EVM)** based blockchain on that Subnet through your Node.js application using **AvalancheJS**. AvalancheJS is a JavaScript library that allows you to issue commands to the Avalanche node APIs without worrying about transaction serialization, signing it with the keys etc.

Requirements

- [Go](#) version >= 1.19.6
- [NodeJS](#) Node >= 10.16 and npm >= 5.6

Project Structure

Open the terminal and make a new directory at your desired location. We will keep all the binaries and other codes in this folder.

```
mkdir subnet-evm-demo  
cd subnet-evm-demo
```

Setting up AvalancheGo and Subnet-EVM Binaries

Clients interact with Avalanche blockchain by issuing API calls to the nodes running `AvalancheGo`. We have to clone the repository, build its binary, and run it. If we want to deploy our blockchain, we have to put the blockchain's VM binary inside the `build/plugins` directory. Here we will also clone the `subnet-evm` repository, build the VM's binary, and copy it to the AvalancheGo's `build/plugins` directory. Follow the steps below -

Clone AvalancheGo Repository

```
git clone https://github.com/ava-labs/avalanchego  
cd avalanchego
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchego.git
```

You can find more about SSH and how to use it [here](#). :::

Build Binary

Running the below command will create an `avalanchego` binary inside the `build/` directory and will install the Coreth `evm` binary inside the `build/plugins` directory. For more information on running a node, please refer [this](#).

```
./scripts/build.sh
```

Clone Subnet-EVM Repository

Move to the `subnet-evm-demo` directory, and clone the repository.

```
git clone https://github.com/ava-labs/subnet-evm  
cd subnet-evm
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/subnet-evm.git
```

You can find more about SSH and how to use it [here](#). :::

Build Binary and Copy it to AvalancheGo Plugins

Now run the following command to build the VM's binary inside the `build/` directory, named as `srEXiWaHuhNyGwPUi444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy`. It is the ID for this VM and corresponds to the string "**Subnet-EVM**" zero-extended in a 32-byte array and encoded in CB58. Then copy it to AvalancheGo's `build/plugins` directory.

```
./scripts/build.sh build/srEXiWaHuhNyGwPUi444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy  
cp build/srEXiWaHuhNyGwPUi444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy ..//avalanchego/build/plugins
```

You can directly build the `subnet-evm` binary inside the `plugins` folder as well, by passing its location as the argument.

Setting up Local Avalanche Network

For the development purpose, we can use [Avalanche Network Runner \(ANR\)](#). It helps us in simulating the actual network. For this tutorial, we will be installing ANR binary and will interact with the network through RPCs.

Install ANR Binary

The following command will install the ANR binary inside `~/bin`. Please make sure that you have the `~/bin` path set in the `$PATH` environment variable, otherwise, you will not be able to run the binary unless you specify its location in each command.

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-network-runner/main/scripts/install.sh | sh -s
```

Start RPC Server

Run the following command to start the RPC server. This will help us in deploying our local cluster of validating nodes. Keep this tab open and run the subsequent commands in the new terminal (or tab).

```
avalanche-network-runner server \  
--port=:8080 \  
--grpc-gateway-port=:8081"
```

Start 5 Node Cluster

Run the following command to start a network cluster of 5 validating nodes, all running the AvalancheGo's binary and have the plugins of the `subnet-evm` VM. Put the `avalanchego` binary location as per your requirement.

```
avalanche-network-runner control start \  
--endpoint="0.0.0.0:8080" \  
--avalanchego-path ${HOME}/subnet-evm-demo/avalanchego/build/avalanchego
```

You can see how the network is set up, and finally the node information at last (within 10-15 seconds), by viewing the logs in the previous tab. Now you have the local simulation of the Avalanche network, with 5 validating nodes.

```
node1: node ID "NodeID-7Xhw2mDxuds44j42TCB6U5579esbSt3Lg", URI "http://localhost:48607"  
node2: node ID "NodeID-MFrZFVCPv5iCn6M9K6XduxGTYp891xX2", URI "http://localhost:27236"  
node3: node ID "NodeID-NFBbbJ4qCmNaCzeW7sxErhvWqvEQMnYcN", URI "http://localhost:58800"  
node4: node ID "NodeID-GWPcbFJZFFzreETSwjPimz846mXEKctu", URI "http://localhost:65011"  
node5: node ID "NodeID-P7oB2McjBGgW2NXXWVYjV8JEDFoW9xD5", URI "http://localhost:12023"
```

You can view the `URIs` of the nodes using the following command.

```
avalanche-network-runner control uris \  
--endpoint="0.0.0.0:8080"
```

For the demonstration purpose, let us choose `node1` as our subject node for making requests and finally making it a validator on our Subnet. Make sure to copy its `URI` and `PORT`, as we will need that later.

ANR also provides us with the funded account with the following credentials.

```
P-Chain Address 1: P-custom18jma8ppw3nhx5r4ap8clazz0dps7rv5u9xde7p  
P-Chain Address 1 Key: PrivateKey-ewoqjP7PxY4yr3iLTpLisriqt94hdyDFNgchSxGGztUrTxtNN
```

We will be using these keys for signing transactions and as the controller of Subnets.

Setting up Node.js Project

Make a new folder for the Node.js project inside the `subnet-evm-demo` directory, so that, the project structure would look like this.

```
$HOME  
|_subnet-evm-demo  
    |_avalanchego  
    |_subnet-evm  
    |_avalanche-network-runner  
    |_subnet-evm-js
```

There is no hard and fast rule to have the same project structure as demonstrated above. You can clone these repositories anywhere you want. You just have to run the commands accordingly to build binaries, copy VMs to `avalanchego/build/plugins/`, run Avalanche Network Runner with AvalancheGo's binary etc.

Installing Dependencies

Here, `subnet-evm-js` is our Node.js project folder. Move to the project directory and install the following dependencies.

- avalanche (3.13.3 or above)
- dotenv
- yargs

```
npm install --save avalanche dotenv yargs
```

Configuration and Other Details

Make a `config.js` file and store the following information, about the node and its respective URI.

```
require("dotenv").config()  
  
module.exports = {  
  protocol: "http",  
  ip: "0.0.0.0",  
  port: 14760,  
  networkID: 1337,  
  privKey: process.env.PRIVATEKEY,  
}
```

We have `networkID: 1337` for the local network. Mainnet has `43114`, and Fuji has `43113`. Put the port here, that you have copied earlier from the ANR's output. Rest all should remain the same. Here we are also accessing `PRIVATEKEY` from the `.env` file. So make sure to include your funded private key in the `.env` file which was provided by ANR.

```
PRIVATEKEY="PrivateKey-ewoqjP7PxY4yr3iLTpLisriqt94hdyDFNgchSxGGztUrTxtNN"
```

Always put secret information like the `.env` file restricted to yourself only and refrain from committing it to git by including it in the `.gitignore` file.

Import Libraries and Setup Instances for Avalanche APIs

This code will serve as the helper function for all other functions spread over different files. It will instantiate all the necessary Avalanche APIs using AvalancheJS and export them for other files to use it. Other files can simply import and re-use. Make a new file `importAPI.js` and paste the following code inside it.

```
const { Avalanche, BinTools, BN } = require("avalanche")  
  
// Importing node details and Private key from the config file.  
const { ip, port, protocol, networkID, privKey } = require("./config.js")  
  
// For encoding and decoding to CB58 and buffers.  
const bintools = BinTools.getInstance()  
  
// Avalanche instance  
const avalanche = new Avalanche(ip, port, protocol, networkID)  
  
// Platform and Info API  
const platform = avalanche.PChain()  
const info = avalanche.Info()  
  
// Keychain for signing transactions  
const pKeyChain = platform.keyChain()  
pKeyChain.importKey(privKey)
```

```
const pAddressStrings = pKeyChain.getAddressStrings()

// UTXOs for spending unspent outputs
const utxoSet = async () => {
  const platformUTXOs = await platform.getUTXOs(pAddressStrings)
  return platformUTXOs.utxos
}

// Exporting these for other files to use
module.exports = {
  platform,
  info,
  pKeyChain,
  pAddressStrings,
  bintools,
  utxoSet,
  BN,
}
```

Genesis Data

Each blockchain has some genesis state when it's created. Each VM defines the format and semantics of its genesis data. We will be using the default genesis data provided by `subnet-evm`. You can also find it inside the `networks/11111/` folder of the `subnet-evm` repository or simply copy and paste the following data inside the `genesis.json` file of the project folder. (Note that fields `airdropHash` and `airdropAmount` have been removed.)

This file will be responsible for setting up of origin state of the chain, like initial balance allocations of the native asset, transaction fees, restricting smart contract access, etc. You should look into 2 major parameters here in the genesis file -

```
"alloc": {
    "d109c2fcfc7fE7AE9ccdB37529E50772053Eb7EE": {
        "balance": "0x52B7D2DCC80CD2E4000000"
    }
}
```

Put your Ethereum derived hexadecimal address (without `0x`) like what you have on the C-Chain. This address will receive the associated balance. Make sure to put your controlled address here, as without this you cannot distribute tokens to users for interacting with your chain (tokens are required for paying fees for the transactions). For development purposes, you can create a new address on MetaMask and use that here.

```
"chainId": 1111
```

Put a unique number as ChainID for your chain. Conflicting ChainIDs can cause problems. You can read more about all of these parameters [here](#).

Creating Subnet

Let's make a file for creating a new Subnet by issuing `buildCreateSubnetTx` on AvalancheJS' `platform` API. Two of the interesting arguments of this function are `subnet-owner` and `threshold`.

Subnet owners control the Subnets by creating signed transactions for adding validators, creating new chains, etc. Whereas the threshold defines the minimum signature required for approving on behalf of all Subnet owners. By default it is 1, so will not pass any argument for that, as we have only one Subnet owner. Code is well commented for you to understand.

```
const {
  platform,
  pKeyChain,
  pAddressStrings,
  utxoSet,
} = require("./importAPI.js")

async function createSubnet() {
  // Creating unsigned tx
  const unsignedTx = await platform.buildCreateSubnetTx(
    await utxoSet(), // set of utxos this tx will consume
    pAddressStrings, // from
    pAddressStrings, // change address
    pAddressStrings // Subnet owners' address array
  )

  // signing unsigned tx with pKeyChain
  const tx = unsignedTx.sign(pKeyChain)

  // issuing tx
  const txId = await platform.issueTx(tx)
  console.log("Tx ID: ", txId)
}

createSubnet()
```

Make sure to keep the `txId` you received for this transaction. Once this transaction is accepted, a new Subnet with the same ID will be created. You can run this program now with the following command.

```
node createSubnet.js
```

Adding Subnet Validator

The newly created Subnet requires validators to validate the transactions on the Subnet's every blockchain. Now we will write the code in `addSubnetValidator.js` to add validators to the Subnet. Only transactions signed by the threshold (here 1) number of Subnet owners will be accepted to add validators. The Subnet owners which will sign this transaction is passed as the `subnetAuth` parameter. It is an array of indices, representing the Subnet owners from the array of addresses that we passed earlier in the `createSubnetTx()`.

The arguments for the AvalancheJS API call for `buildAddSubnetValidatorTx()` is explained with the help of comments. All the transaction calls of AvalancheJS starting with `build` will return an unsigned transaction. We then have to sign it with our key chain and issue the signed transaction to the network.

```
const args = require("yargs").argv
const {
  platform,
  info,
  pKeyChain,
  pAddressStrings,
  utxoSet,
  BN,
} = require("./importAPI.js")

async function addSubnetValidator() {
  let {
    nodeID = await info.getNodeID(),
    startTime,
    endTime,
    weight = 20,
    subnetID,
  }
```

```

} = args

const pAddresses = pKeyChain.getAddresses()

// Creating Subnet auth
const subnetAuth = [[0, pAddresses[0]]]

// Creating unsigned tx
const unsignedTx = await platform.buildAddSubnetValidatorTx(
  await utxoSet(), // set of utxos this tx will consume
  pAddressStrings, // from
  pAddressStrings, // change
  nodeID, // node id of the validator
  new BN(startTime), // timestamp after which validation starts
  new BN(endTime), // timestamp after which validation ends
  new BN(weight), // weight of the validator
  subnetID, // Subnet id for validation
  undefined, // memo
  undefined, // asof
  subnetAuth // Subnet owners' address indices signing this tx
)

// signing unsigned tx with pKeyChain
const tx = unsignedTx.sign(pKeyChain)

// issuing tx
const txId = await platform.issueTx(tx)
console.log("Tx ID: ", txId)
}

addSubnetValidator()

```

We have to pass command-line arguments like `nodeID`, `startTime`, `endTime`, `weight` and `subnetID` while calling the command. If we do not pass any `NodeID`, then by default it will use ID corresponding to the URI in the `config.js` by calling the `info.getNodeID()` API from AvalancheJS. Similarly, the default weight will be 20, if not passed. You can run this program now with the following command.

```

node addSubnetValidator.js \
--subnetID <YOUR_SUBNET_ID> \
--startTime $(date -v +5M +%s) \
--endTime $(date -v +14d +%s)

```

We will keep the start time 5 minutes later than the current time. This `$(date -v +5M +%s)` will help to achieve the same. But you can put any timestamp in seconds there, given it is 20s later than the current time.

Tracking a Subnet from the Node

Subnet owners can add any node to their Subnet. That doesn't mean the nodes start validating their Subnet without any consent. If a node wants to validate the newly added Subnet, then it must restart its `avalanchego` binary with the new Subnet being tracked.

```

avalanche-network-runner control restart-node \
--request-timeout=3m \
--endpoint="0.0.0.0:8080" \
--node-name node1 \
--avalanchego-path ${HOME}/subnet-evm-demo/avalanchego/build/avalanchego \
--track-subnets=<SUBNET_ID>

```

Once the node has restarted, it will again be re-assigned to a random API port. We have to update the `config.js` file with the new port.

Creating Blockchain

Once the Subnet setup is complete, Subnet owners can deploy any number of blockchains by building their own VMs or reusing the existing ones. If the VM for the new blockchain is not being used by the Subnet validators, then each node has to place the new VM binary in their `avalanchego/build/plugins/` folder.

Let's write functions to build a new blockchain using the already created `genesis.json` and `subnet-evm` as a blueprint (VM) for this chain. We will write the code in steps. Go through the steps by understanding each function and pasting it in your `createBlockchain.js` file.

Importing Dependencies

Let's import the dependencies by using the following snippet. We are importing `yargs` for reading command-line flags.

```

const args = require("yargs").argv

const genesisJSON = require("./genesis.json")
const {
  platform,
  pKeyChain,

```

```

pAddressStrings,
bintools,
utxoSet,
} = require("./importAPI")

```

Decoding CB58 vmID to String

We have used the utility function for decoding `vmName` from the `vmID`. `vmID` is zero-extended in a 32-byte array and encoded in CB58 from a string. Paste the function shown below.

```

// Returns string representing vmName of the provided vmID
function convertCB58ToString(cb58Str) {
  const buff = bintools.cb58Decode(cb58Str)
  return buff.toString()
}

```

Create the Blockchain

Now we will work upon the `createBlockchain()` function. This function takes 3-4 command-line flags as its input. The user must provide `subnetID` and `chainName` flag. The third argument could be either `vmID` or `vmName`. Either one of them must be provided with the flags. Chain name is the name of blockchain you want to create with the provided `vmID` or `vmName`. The `vmID` must be the same as what we have created the `subnet-evm` binary with.

```

// Creating blockchain with the subnetID, chain name and vmID (CB58 encoded VM name)
async function createBlockchain() {
  const { subnetID, chainName } = args

  // Generating vmName if only vmID is provided, else assigning args.vmID
  const vmName =
    typeof args.vmName !== "undefined"
      ? args.vmName
      : convertCB58ToString(args.vmID)

  // Getting CB58 encoded bytes of genesis
  genesisBytes = JSON.stringify(genesisJSON)

  const pAddresses = pKeyChain.getAddresses()

  // Creating Subnet auth
  const subnetAuth = [[0, pAddresses[0]]]

  // Creating unsigned tx
  const unsignedTx = await platform.buildCreateChainTx(
    await utxoSet(), // set of utxos this tx is consuming
    pAddressStrings, // from
    pAddressStrings, // change
    subnetID, // id of Subnet on which chain is being created
    chainName, // Name of blockchain
    vmName, // Name of the VM this chain is referencing
    [], // Array of feature extensions
    genesisBytes, // Stringified genesis JSON file
    undefined, // memo
    undefined, // asof
    subnetAuth // Subnet owners' address indices signing this tx
  )

  // Signing unsigned tx with pKeyChain
  const tx = unsignedTx.sign(pKeyChain)

  // Issuing tx
  const txId = await platform.issueTx(tx)
  console.log("Create chain transaction ID: ", txId)
}

```

The code above is self-explanatory -

- Processing command-line flags to constants - `subnetID`, `chainName` and `vmID`.
- Building stringified JSON from `genesis.json`
- Creating Subnet auth array.
- Creating unsigned TX by calling `platform.buildCreateChainTx`.
- Signing and issuing the signed TXs.

Make sure to keep `txID` received by running this code. Once the transaction is committed, the `txID` will be the `blockchainID` or identifier for the newly created chain. You can run this program now with the following command.

```

node createBlockchain.js \
--subnetID <YOUR_SUBNET_ID> \

```

```
--chainName <CUSTOM_CHAIN_NAME> \
--vmName subnetevm
```

Creating a new chain will take few seconds. You can also view the logs on the Avalanche Network Runner tab of the terminal.

Interacting with the New Blockchain with MetaMask

We have created the new Subnet, deployed a new blockchain using the `subnet-evm`, and finally added a validator to this Subnet, for validating different chains. Now it's time to interact with the new chain. You can follow this [part](#) in our docs, to learn, how you can set up your MetaMask to interact with this chain. You can send tokens, create smart contracts, and do everything that you can do on C-Chain.

Deploy a Permissioned Subnet on Testnet

:::note

This document describes how to use the new Avalanche-CLI to deploy a Subnet on `Fuji`.

:::

After trying out a Subnet on a local box by following [this tutorial](#), next step is to try it out on `Fuji` Testnet.

In this article, it's shown how to do the following on `Fuji` Testnet.

- Create a Subnet.
- Deploy a virtual machine based on Subnet-EVM.
- Add a node as a validator to the Subnet.
- Join a node to the newly created Subnet.

All IDs in this article are for illustration purposes. They can be different in your own run-through of this tutorial.

Prerequisites

- 1+ nodes running and fully bootstrapped on `Fuji` Testnet. Check out the section [Nodes](#) on how to run a node and become a validator.
- [Avalanche-CLI](#) installed

Virtual Machine

Avalanche can run multiple blockchains. Each blockchain is an instance of a [Virtual Machine](#), much like an object in an object-oriented language is an instance of a class. That's, the VM defines the behavior of the blockchain.

[Subnet-EVM](#) is the VM that defines the Subnet Contract Chains. Subnet-EVM is a simplified version of [Avalanche C-Chain](#).

This chain implements the Ethereum Virtual Machine and supports Solidity smart contracts as well as most other Ethereum client features.

Fuji Testnet

For this tutorial, it's recommended that you follow [Run an Avalanche Node Manually](#) and this step below particularly to start your node on `Fuji`:

To connect to the `Fuji` Testnet instead of the main net, use argument `--network-id=Fuji`

Also it's worth pointing out that [it only needs 1 AVAX to become a validator on the Fuji Testnet](#) and you can get the test token from the [faucet](#).

To get the NodeID of this `Fuji` node, call the following curl command to [info.getNodeID](#):

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

The response should look something like:

```
{
  "jsonrpc": "2.0",
  "result": {
    "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD"
  },
  "id": 1
}
```

That portion that says, `NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD` is the NodeID, the entire thing. The user is going to need this ID in the later section when calling [addValidator](#).

:::info

With more data on `Fuji`, it may take a while to bootstrap `Fuji` Testnet from scratch. You can use [State-Sync](#) to shorten the time for bootstrapping.

:::

Avalanche-CLI

If not yet installed, install Avalanche-CLI following the tutorial at [Avalanche-CLI installation](#).

Private Key

All commands which issue a transaction require either a private key loaded into the tool, or a connected ledger device.

This tutorial focuses on stored key usage and leave ledger operation details for the `Mainnet` deploy one, as `Mainnet` operations requires ledger usage, while for `Fuji` it's optional.

Avalanche-CLI supports the following key operations:

- create
 - delete
 - export
 - list

:::warning

You should only use the private key created for this tutorial for testing operations on `Fuji` or other testnets. Don't use this key on `Mainnet`. CLI is going to store the key on your file system. Whoever gets access to that key is going to have access to all funds secured by that private key. To deploy to `Mainnet`, follow [this tutorial](#).

1

Run `create` if you don't have any private key available yet. You can create multiple named keys. Each command requiring a key is going to therefore require the appropriate key name you want to use.

```
avalanche key create mytestkey
```

This is going to generate a new key named `mytestkey`. The command is going to then also print addresses associated with the key:

```
Generating new key...
Key created
+-----+
| KEY NAME |          CHAIN          |          ADDRESS          | NETWORK |
+-----+
| mytestkey | C-Chain (Ethereum hex format) | 0x86BB07a534ADF43786ECA5Dd34A97e3F96927e4F | All     |
+-----+
|           | P-Chain (Bech32 format)      | P-custom1a3azftqvgygc4tlqsdvd82wks2u7nx85rg7v8ta | Local Network |
+-----+
|           |                           +-----+
|           | P-fujil1a3azftqvgygc4tlqsdvd82wks2u7nx85rhk6zqh | Fuji    |
+-----+
```

You may use the C-Chain address (0x86BB07a534ADF43786ECA5Dd34A97e3F96927e4F) to fund your key from the [faucet](#). The command also prints P-Chain addresses for both the default local network and Fuji . The latter (P-fujila3azftqvygc4tlgsdvd8wks2u7nx85rkhk6zqh) is the one needed for this tutorial.

The `delete` command of course deletes a private key:

```
avalanche key delete mytestkey
```

Be careful though to always have a key available for commands involving transactions.

The `export` command is going to **print your private key** in hex format to stdout.

this key is intentionally modified.

You can also **import** a key by using the `--file` flag with a path argument and also providing a name to it:

```
avalanche key create othertest --file /tmp/test.pk  
Loading user key...  
Key loaded
```

Finally, the `list` command is going to list all your keys in your system and their associated addresses (CLI stores the keys in a special directory on your file system, tampering with the directory is going to result in malfunction of the tool).

avalanche key list				
KEY NAME	CHAIN	ADDRESS	NETWORK	
othertest	C-Chain (Ethereum hex format)	0x36c83263e33f9e87EB98D3feB54a01E35a3Fa735	All	
	P-Chain (Bech32 format)	P-custom1n5n4h99j3nx8hdrv50v81l7aldm38nap6rh42	Local Network	

```

+      +
|      |          +-----+
|      |          | P-fujiln5n4h99j3nx8hdrv50v8ll7alldm383na7j4j7q | Fuji   |
+-----+-----+-----+
| mytestkey | C-Chain (Ethereum hex format) | 0x86BB07a534ADF43786ECA5Dd34A97e3F96927e4F | Al1    |
+-----+-----+-----+
|      | P-Chain (Bech32 format)           | P-custom1a3azftqvyygc4tlqsdvd82wks2u7nx85rg7v8ta | Local Network |
+-----+-----+-----+
|      | P-fujila3azftqvyygc4tlqsdvd82wks2u7nx85rkh6zqh | Fuji   |
+-----+-----+-----+

```

Funding the Key

:::danger

Do these steps only to follow this tutorial for `Fuji` addresses. To access the wallet for `Mainnet`, the use of a ledger device is strongly recommended.

:::

1. A newly created key has no funds on it. Send funds via transfer to its correspondent addresses if you already have funds on a different address, or get it from the faucet at <https://faucet.avax.network> using your **C-Chain address**.
2. **Export** your key via the `avalanche key export` command, then paste the output when selecting "Private key" while accessing the [web wallet](#). (Private Key is the first option on the [web wallet](#)).
3. Move the test funds from the C-Chain to the P-Chain by clicking on the `Cross Chain` on the left side of the web wallet (find more details on [this tutorial](#)).

After following these 3 steps, your test key should now have a balance on the P-Chain on `Fuji Testnet`.

Create an EVM Subnet

Creating a Subnet with `Avalanche-CLI` for `Fuji` works the same way as with a [local network](#). In fact, the `create` commands only creates a specification of your Subnet on the local file system. Afterwards the Subnet needs to be *deployed*. This allows to reuse configs, by creating the config with the `create` command, then first deploying to a local network and successively to `Fuji` - and eventually to `Mainnet`.

To create an EVM Subnet, run the `subnet create` command with a name of your choice:

```
avalanche subnet create testsubnet
```

This is going to start a series of prompts to customize your EVM Subnet to your needs. Most prompts have some validation to reduce issues due to invalid input. The first prompt asks for the type of the virtual machine (see [Virtual Machine](#)).

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose your VM:
▶ SubnetEVM
Custom
```

As you want to create an EVM Subnet, just accept the default `Subnet-EVM`. Next, CLI asks for the ChainID. You should provide your own ID. Check [chainlist.org](#) to see if the value you'd like is already in use.

```
✓ Subnet-EVM
creating subnet testsubnet
Enter your subnet's ChainId. It can be any positive integer.
ChainId: 3333
```

Now, provide a symbol of your choice for the token of this EVM:

```
Select a symbol for your subnet's native token
Token symbol: TST
```

At this point, CLI prompts the user for the fee structure of the Subnet, so that he can tune the fees to the needs:

```
Use the arrow keys to navigate: ↑ ↓ ← →
? How would you like to set fees:
▶ Low disk use / Low Throughput 1.5 mil gas/s (C-Chain's setting)
  Medium disk use / Medium Throughput 2 mil gas/s
  High disk use / High Throughput 5 mil gas/s
  Customize fee config
  Go back to previous step
```

You can navigate with the arrow keys to select the suitable setting. Use `Low disk use / Low Throughput 1.5 mil gas/s` for this tutorial.

The next question is about the airdrop:

```
✓ Low disk use / Low Throughput 1.5 mil gas/s
Use the arrow keys to navigate: ↑ ↓ ← →
```

```
? How would you like to distribute funds:
▶ Airdrop 1 million tokens to the default address (do not use in production)
  Customize your airdrop
  Go back to previous step
```

You can accept the default -again, NOT for production-, or customize your airdrop. In the latter case the wizard would continue. Assume the default here.

The final question is asking for precompiles. Precompiles are powerful customizations of your EVM. Read about them at [precompiles](#).

```
✓ Airdrop 1 million tokens to the default address (do not use in production)
Use the arrow keys to navigate: ↑ ↓ ← →
? Advanced: Would you like to add a custom precompile to modify the EVM?:
▶ No
  Yes
  Go back to previous step
```

For this tutorial, assume the simple case of no additional precompile. This finalizes the prompt sequence and the command exits:

```
✓ No
Successfully created genesis
```

It's possible to end the process with Ctrl-C at any time.

At this point, CLI creates the specification of the new Subnet on disk, but isn't deployed yet.

Print the specification to disk by running the `describe` command:

```
avalanche subnet describe testsubnet
_____
|   \   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+-----+
|     PARAMETER      |           VALUE          |
+-----+
| Subnet Name       | testsubnet          |
+-----+
| ChainID          | 3333              |
+-----+
| Token Name        | TST                |
+-----+
| VM ID            | tGBFM2SXkAdNsqb3SaFZZWMNdzjjFEUKteheTa4diUwnfQyu |
+-----+
| Fuji SubnetID    | XTK7AM2Pw5A4cCtQ3rTuggbeLCU9mVixML3YwwLYUJ4WXN2Kt |
+-----+
| Fuji BlockchainID | 5ce2WhnyeMELzg9UtpfCDGNwRa2AzMzRhBWfTqmFuiXPWE4TR |
+-----+
| Local Network SubnetID | 2CZP2ndbQnZxTzGuZjPrJAm5b4s2K2Bcjh8NqNoymi8NZMLYQk |
+-----+
| Local Network BlockchainID | oaCmwvn8FDuv8QjeTozGpHeczk1Kv2651j2jh6sRlnraGwVW |
+-----+
_____
|   \   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+-----+
|     GAS PARAMETER    |     VALUE      |
+-----+
| GasLimit           | 15000000 |
+-----+
| MinBaseFee         | 25000000000 |
+-----+
| TargetGas (per 10s) | 20000000 |
+-----+
| BaseFeeChangeDenominator | 36 |
+-----+
| MinBlockGasCost    | 0 |
+-----+
| MaxBlockGasCost    | 1000000 |
+-----+
```

```

| TargetBlockRate | 2 |
+-----+
| BlockGasCostStep | 200000 |
+-----+



+-----+
| ADDRESS | AIRDROP AMOUNT (10^18) | AIRDROP AMOUNT (WEI) |
+-----+
| 0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC | 1000000 | 10000000000000000000000000000000 |
+-----+

```

```

+-----+
| ADDRESS | AIRDROP AMOUNT (10^18) | AIRDROP AMOUNT (WEI) |
+-----+
| 0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC | 1000000 | 10000000000000000000000000000000 |
+-----+

```

No precompiles set

Also you can list the available Subnets:

```

avalanche subnet list
go run main.go subnet list
+-----+
| SUBNET | CHAIN | CHAIN ID | VM ID | TYPE | FROM REPO |
+-----+
| testsubnet | testsubnet | 3333 | tGBBrM2SXkAdNsqzb3SaFZZWMNdzjjFEUKteheTa4dhUwnfQyu | Subnet-EVM | false |
+-----+

```

List deployed information:

```

avalanche subnet list --deployed
go run main.go subnet list --deployed
+-----+
| SUBNET | CHAIN | VM ID | LOCAL NETWORK | FUJI |
| (TESTNET) | MAINNET | +-----+
+-----+
| testsubnet | testsubnet | tGBBrM2SXkAdNsqzb3SaFZZWMNdzjjFEUKteheTa4dhUwnfQyu | Yes | SubnetID: XTK7AM2Pw5A4cCtQ3rTuggbeLCU9mVixML3YwwLYUJ4WVN2Kt | No | +-----+
+-----+
| +-----+ | +-----+ | +-----+ | +-----+ | +-----+ | +-----+ |
| 5ce2WhnyeMELzg9UtfpCDGNwRa2AzMzRhBWFtqmfuiXPWE4TR | No | +-----+ | BlockchainID: +-----+
+-----+

```

Deploy the Subnet

To deploy the new Subnet, run

```
avalanche subnet deploy testsubnet
```

This is going to start a new prompt series.

```

Use the arrow keys to navigate: ↑ → ←
? Choose a network to deploy on:
▶ Local Network
Fuji
Mainnet

```

This tutorial is about deploying to `Fuji`, so navigate with the arrow keys to `Fuji` and hit enter. The user is then asked to provide which private key to use for the deployment. The deployment basically consists in running a [createSubnet transaction](#). Therefore the key needs to have funds.

Also, this tutorial assumes that a node is up running, fully bootstrapped on `Fuji`, and runs from the **same** box.

```
✓ Fuji
Deploying [testsubnet] to Fuji
Use the arrow keys to navigate: ↑ ↓ ← →
? Which private key should be used to issue the transaction?:
  test
▶ mytestkey
```

Subnets are currently permissioned only. Therefore, the process now requires the user to provide *which keys can control the Subnet*. CLI prompts the user to provide one or more **P-Chain addresses**. Only the keys corresponding to these addresses are going to be able to add or remove validators. Make sure to provide **Fuji P-Chain addresses - P-Fuji.... -**.

```
Configure which addresses may add new validators to the subnet.
These addresses are known as your control keys. You are going to also
set how many control keys are required to add a validator.
Use the arrow keys to navigate: ↑ ↓ ← →
? Set control keys:
  ▶ Add control key
  Done
  Cancel
```

Enter at `Add control key` and provide at least one key. You can enter multiple addresses, just use one here. When finishing, hit `Done`. (The address provided here is intentionally invalid. The address has a checksum and the tool is going to make sure it's a valid address).

```
✓ Add control key
Enter P-Chain address (Ex: `P-...`): P-fujilvaaaaaaaaaaaaaaaaaaaaaaaaaaaaasz
Use the arrow keys to navigate: ↑ ↓ ← →
? Set control keys:
  ▶ Add control key
  ▶ Done
  Cancel
```

Finally, there is a need to define the threshold of how many keys to require for a change to be valid -there is some input validation-. For example, if the is one control key, as preceding, just enter 1. The threshold could be arbitrary depending on the needs, for example 2 of 4 addresses, 1 of 3, 3 of 5, etc., but currently this tool only works if *the private key used here owns at least one control key and the threshold is 1*.

```
✓ Enter required number of control key signatures to add a validator: 1
```

Here the wizard completes, and CLI attempts the transaction.

If the private key isn't funded or doesn't have enough funds, the error message is going to be:

```
Error: insufficient funds: provided UTXOs need 100000000 more units of asset "U8iRqJoiJm8xZHaacmvYyZVwqQx6uDNTQeP3CQ6fcgQk3JqnK"
```

If the private key has funds, but the **control key** is incorrect (not controlled by the private key), the CLI is going to create the Subnet, but *not the blockchain*:

```
Subnet has been created with ID: 2EkPnvnDiLgudnf8NjtxaNcVFtdAAnUPvaoNBrc9WG5tNmfaK. Now creating blockchain...
Error: insufficient authorization
```

Therefore the user needs to provide a control key which he has indeed control of, and then it succeeds. The output (assuming the node is running on `localhost` and the API port set to standard `9650`) is going to look something like this:

```
Subnet has been created with ID: 2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H. Now creating blockchain...
Endpoint for blockchain "2XDnKyAEr1RhhWpTpMXqrjeejn23vETmDykVzkb4PrUlQjewh" with VM ID
"tGBrMADEsojmu5Et9CpbGcrmVf9fiAJtzM5ZJ3YDj5JTu2gw":
http://127.0.0.1:9650/ext/bc/2XDnKyAEr1RhhWpTpMXqrjeejn23vETmDykVzkb4PrUlQjewh/rpc
```

Well done. You have just created your own Subnet with your own Subnet-EVM running on `Fuji`.

To get your new Subnet information, visit the [Avalanche Subnet Explorer](#). The search best works by blockchain ID, so in this example, enter `2XDnKyAEr1RhhWpTpMXqrjeejn23vETmDykVzkb4PrUlQjewh` into the search box and you should see your shiny new blockchain information.

Add a Validator

:::info

Adding a validator on a Subnet requires that the node is already a validator on the primary network, which means that your node has **fully bootstrapped**.

See [here](#) on how to become a validator.

:::

This new Subnet is cool - but it doesn't have any dedicated validators yet. Add one by running the `addValidator` command and adding the name of the Subnet. To be clear, this does *not start or run* a validator, it only whitelists the node as a recognized validator on the Subnet.

```
avalanche subnet addValidator testsubnet
```

As this operation involves a new [transaction](#), it's needed to tell the tool which private key to use:

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Which private key should be used to issue the transaction?:
  test
▶ mytestkey
```

Choose [Fuji](#):

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to deploy on. This command only supports Fuji currently.:
  ▶ Fuji
  Mainnet
```

Now use the **NodeID** of the new validator from the very beginning of this tutorial. For best results make sure the validator is running and synced.

```
What is the NodeID of the validator you'd like to whitelist?: NodeID-BFa1paAAAAAAAAAAAAAAAQGjPhUy
```

-this ID is intentionally modified-

The next question requires a bit of thinking. A validator has a weight, which defines how often consensus selects it for decision making. You should think ahead of how many validators you want initially to identify a good value here. The range is 1 to 100, but the minimum for a Subnet without any validators yet is 20. The structure is a bit described at [addSubnetValidator](#) under the `weight` section.

Just select 30 for this one:

```
Use the arrow keys to navigate: ↑ ↓ ← →
? What stake weight would you like to assign to the validator?:
  Default (20)
  ▶ Custom
```

```
✓ What stake weight would you like to assign to the validator?: 30
```

Then specify when the validator is going to start validating. The time must be in the future. Custom option is going to require to enter a specific date in `YYYY-MM-DD HH:MM:SS` format. Just take the default this time:

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Start time:
  ▶ Start in one minute
  Custom
```

:::warning

If the `join` command isn't successfully completed before this time elapses and the validator's stake weight is >20% of the Subnet, the Subnet may have down time.

:::

Finally, specify how long it's going to be validating:

```
✓ Start in one minute
Use the arrow keys to navigate: ↑ ↓ ← →
? How long should your validator validate for?:
  ▶ Until primary network validator expires
  Custom
```

If choosing `Custom` here, the user must enter a **duration**, which is a time span expressed in hours. For example, could say `200 days = 24 * 200 = 4800 hours`

```
✓ How long should this validator be validating? Enter a duration, e.g. 8760h: 4800h
```

CLI shows an actual date of when that's now:

```
? Your validator is going to finish staking by 2023-02-13 12:26:55:
  ▶ Yes
  No
```

Confirm if correct. At this point the prompt series is complete and CLI attempts the transaction:

```
NodeID: NodeID-BFa1padLXBj7VHa2JYvYGzcTBFQGjPhUy
Network: Fuji
Start time: 2022-07-28 12:26:55
End time: 2023-02-13 12:26:55
Weight: 30
Inputs complete, issuing transaction to add the provided validator information...
```

This might take a couple of seconds, and if successful, it's going to print:

```
Transaction successful, transaction ID :EhZh8PvQyqA9xggxn6EsdemXMnWKyy839NzEJ5DHExTBiXbjV
```

This means the node is now a validator on the given Subnet on `Fuji` !

Join a Subnet

You might already have a running validator which you want to add to a specific Subnet. For this run the `join` command. This is a bit of a special command. The `join` command is going to either just *print the required instructions* for your already running node or is going to attempt at configuring a config file the user provides.

First network selection:

```
avalanche subnet join testsubnet
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to validate on (this command only supports public networks):
▶ Fuji
Mainnet
```

The [Deploy the Subnet](#) section shows that a Subnet is permissioned via a set of keys. Therefore not any node is suitable as validator to the Subnet. A holder of a control key *must* call [Subnet addValidator](#) first in order to allow the node to validate the Subnet. So the tool allows the user now to verify if the node has already been permissioned -"whitelisted"- to be a validator for this Subnet -by calling an API in the background-.

```
Would you like to check if your node is allowed to join this subnet?
If not, the subnet's control key holder must call avalanche subnet
addValidator with your NodeID.
Use the arrow keys to navigate: ↑ ↓ ← →
? Check whitelist?:
    Yes
    ▶ No
```

The default is `Yes` but just choose `No` here to speed up things, assuming the node is already whitelisted.

There are now two choices possible: automatic and Manual configuration. As mentioned earlier, "Automatic" is going to attempt at editing a config file and setting up your plugin directory, while "Manual" is going to just print the required config to the screen. See what "Automatic" does:

```
✓ Automatic
✓ Path to your existing config file (or where it's going to be generated): config.json
```

Provide a path to a config file. If executing this command on the box where your validator is running, then you could point this to the actually used config file, for example `/etc/avalanchego/config.json` - just make sure the tool has **write** access to the file. Or you could just copy the file later. In any case, the tool is going to either try to edit the existing file specified by the given path, or create a new file. Again, set write permissions.

Next, provide the plugin directory. The beginning of this tutorial contains VMs description [Virtual Machine](#). Each VM runs its own plugin, therefore AvalancheGo needs to be able to access the correspondent plugin binary. As this is the `join` command, which doesn't know yet about the plugin, there is a need to provide the directory where the plugin resides. Make sure to provide the location for your case:

```
✓ Path to your avalanchego plugin dir (likely avalanchego/build/plugins): /home/user/go/src/github.com/avalanchego/avalanchego/build/plugins
```

The tool doesn't know where exactly it's located so it requires the full path. With the path given, it's going to copy the VM binary to the provided location:

```
✓ Path to your avalanchego plugin dir (likely avalanchego/build/plugins): /home/user/go/src/github.com/avalanchego/avalanchego/build/plugins
VM binary written to /home/user/go/src/github.com/avalanchego/avalanchego/build/plugins/tGBrMADESojmu5Et9CpbGCrmVf9fiAJtZM5ZJ3YVDj5JTu2qw
This is going to edit your existing config file. This edit is nondestructive,
but it's always good to have a backup.
Use the arrow keys to navigate: ↑ ↓ ← →
? Proceed?:
    Yes
    ▶ No
```

Hitting `Yes` is going to attempt at writing the config file:

✓ Yes
The config file has been edited. To use it, make sure to start the node with the '--config-file' option, e.g.
`./build/avalanchego --config-file config.json`
(using your binary location). The node has to be restarted for the changes to take effect.

It's required to restart the node.

If choosing "Manual" instead, the tool is going to just print *instructions*. The user is going to have to follow these instructions and apply them to the node. Note that the IDs for the VM and Subnets is going to be different in your case.

✓ Manual
To setup your node, you must `do` two things:
1. Add your VM binary to your node's plugin directory
2. Update your node config to start validating the subnet
To add the VM to your plugin directory, copy or scp from `/tmp/tGBrMADESojmu5Et9CpbGCrmVf9fiAJtZM5ZJ3YVDj5JTu2qw`
If you installed `avalanchego` manually, your plugin directory is likely `avalanchego/build/plugins`.
If you start your node from the command line WITHOUT a config file (e.g. via command line or systemd script), add the following flag to your node's startup `command`:
`--track-subnets=2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H`
(if the node already has a `track-subnets` config, append the new value by comma-separating it).
For example:
`./build/avalanchego --network-id=Fuji --track-subnets=2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H`
If you start the node via a JSON config file, add this to your config file:
`track-subnets: 2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H`
TIP: Try this `command` with the `--avalanchego-config` flag pointing to your config file, this tool is going to try to update the file automatically (make sure it can write to it).
After you update your config, you are going to need to restart your node `for` the changes to take effect.

Subnet Export

This tool is most useful on the machine where a validator is or is going to be running. In order to allow a VM to run on a different machine, you can export the configuration. Just need to provide a path to where to export the data:

```
avalanche subnet export testsubnet
✓ Enter file path to write export data to: /tmp/testsubnet-export.dat
```

The file is in text format and you shouldn't change it. You can then use it to import the configuration on a different machine.

Subnet Import

To import a VM specification exported in the previous section, just issue the `import` command with the path to the file after having copied the file over:

```
avalanche subnet import /tmp/testsubnet-export.dat
Subnet imported successfully
```

After this the whole Subnet configuration should be available on the target machine:

```
avalanche subnet list
+-----+-----+-----+-----+
|   SUBNET   |   CHAIN   | CHAIN ID |   TYPE   | DEPLOYED |
+-----+-----+-----+-----+
| testsubnet | testsubnet |     3333 | SubnetEVM | No      |
+-----+-----+-----+-----+
```

Appendix

Connect with MetaMask

To connect MetaMask with your blockchain on the new Subnet running on your local computer, you can add a new network on MetaMask with the following values:

```
- Network Name: testsubnet
- RPC URL: <http://127.0.0.1:9650/ext/bc/2XDnKyAEr1RhhWpTpMXqrjeejN23vETmDykvzbk4PrU1fQjewh/rpc>
- Chain ID: 3333
- Symbol: TST
```

:::note

Unless you deploy your Subnet on other nodes, you aren't going to be able to use other nodes, including the public API server <https://api.avax-test.network/>, to connect to MetaMask.

If you want to open up this node for others to access your Subnet, you should set it up properly with `https://node-ip-address` instead of `http://127.0.0.1:9650`, however, it's out of scope for this tutorial on how to do that.

:::

How to Deploy a Subnet on a Local Network

This how-to guide focuses on taking an already created Subnet configuration and deploying it to a local Avalanche network.

Prerequisites

- [Avalanche-CLI installed](#)
- You have [created a Subnet configuration](#)

Deploying Subnets Locally

In the following commands, make sure to substitute the name of your Subnet configuration for `<subnetName>`.

To deploy your Subnet, run

```
avalanche subnet deploy <subnetName>
```

and select `Local Network` to deploy on. Alternatively, you can bypass this prompt by providing the `--local` flag. For example:

```
avalanche subnet deploy <subnetName> --local
```

The command may take a couple minutes to run.

Note: If you run `bash` on your shell and are running Avalanche-CLI on ARM64 on Mac, you will require Rosetta 2 to be able to deploy Subnets locally. You can download Rosetta 2 using `softwareupdate --install-rosetta`.

Results

If all works as expected, the command output should look something like this:

```
> avalanche subnet deploy mySubnet
✓ Local Network
Deploying [mySubnet] to Local Network
Installing subnet-evm-v0.4.3...
subnet-evm-v0.4.3 installation successful
Backend controller started, pid: 93928, output at: /Users/subnet-developer/.avalanche-cli/runs/server_20221122_173138/avalanche-
cli-backend
Installing avalanchego-v1.9.3...
avalanchego-v1.9.3 installation successful
VMs ready.
Starting network...
.....
Blockchain has been deployed. Wait until network acknowledges...
.....
Network ready to use. Local network node endpoints:
+-----+-----+
| NODE | VM | URL |
+-----+-----+
| node2 | mySubnet | http://127.0.0.1:9652/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node3 | mySubnet | http://127.0.0.1:9654/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node4 | mySubnet | http://127.0.0.1:9656/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node5 | mySubnet | http://127.0.0.1:9658/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node1 | mySubnet | http://127.0.0.1:9650/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+

Browser Extension connection details (any node URL from above works):
RPC URL: http://127.0.0.1:9650/ext/bc/SPqou41AAALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Funded address: 0x8db97C7cEcE249c2b98bdC0226Cc4C2A57BF52FC with 1000000 (10^18) - private key:
56289e99c94b6912bfc12adc093c9b51124f0dc54ac7a766b2bc5ccf558d8027
Network name: mySubnet
```

```
Chain ID: 54325
Currency Symbol: TUTORIAL
```

You can use the deployment details to connect to and interact with your Subnet.

To manage the newly deployed local Avalanche network, see [the avalanche network command tree](#).

Deploying Multiple Subnets

You may deploy multiple Subnets concurrently, but you can't deploy the same Subnet multiple times without resetting all deployed Subnet state.

Redeploying the Subnet

To redeploy the Subnet, you first need to wipe the Subnet state. This permanently deletes all data from all locally deployed Subnets. To do so, run

```
avalanche network clean
```

You are now free to redeploy your Subnet with

```
avalanche subnet deploy <subnetName> --local
```

Deploy a Permissioned Subnet on Mainnet

:::warning

Deploying a Subnet to Mainnet has many risks. Doing so safely requires a laser focus on security. This tutorial does its best to point out common pitfalls, but there may be other risks not discussed here.

This tutorial is an educational resource and provides no guarantees that following it results in a secure deployment. Additionally, this tutorial takes some shortcuts that aid the understanding of the deployment process at the expense of security. The text highlights these shortcuts and they shouldn't be used for a production deployment.

:::

After managing a successful Subnet deployment on the [Fuji Testnet](#), you're ready to deploy your Subnet on Mainnet. If you haven't done so, first [Deploy a Subnet on Testnet](#).

This tutorial shows how to do the following on [Mainnet](#).

- Create a Subnet.
- Deploy a virtual machine based on Subnet-EVM.
- Add a node as a validator to the Subnet.
- Join a node to the newly created Subnet.

:::note

All IDs in this article are for illustration purposes only. They are guaranteed to be different in your own run-through of this tutorial.

:::

Prerequisites

- 5+ nodes running and [fully bootstrapped](#) on [Mainnet](#)
- [Avalanche-CLI is installed](#) on each validator node's box
- A [Ledger](#) device
- You've [created a Subnet configuration](#) and fully tested a [Fuji Testnet Subnet deployment](#)

:::warning

Although only one validator is strictly required to run a Subnet, running with less than five validators is extremely dangerous and guarantees network downtime. Plan to support at least five validators in your production network.

:::

Getting Your Mainnet NodeIDs

You need to collect the NodeIDs for each of your validators. This tutorial uses these NodeIDs in several commands.

To get the NodeID of a [Mainnet](#) node, call the [info.getNodeID](#) endpoint. For example:

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeID"
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/info
```

The response should look something like:

```
{
  "jsonrpc": "2.0",
```

```

"result": {
  "nodeID": "NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD"
},
"id": 1
}

```

In the sample response, `NodeID-5mb46qkSBj81k9g9e4VFjGGSbaaSLFRzD` is the `NodeID`. Note that the `NodeID-` prefix is part of the `NodeID`.

Setting up Your Ledger

In the interest of security, all Avalanche-CLI `Mainnet` operations require the use of a connected Ledger device. You must unlock your Ledger and run the Avalanche App. See [How to Use Ledger](#) for help getting set up.

Avalanche-CLI supports the Ledger `Nano X`, `Nano S`, and `Nano S Plus`.

Ledger devices support TX signing for any address inside a sequence automatically generated by the device.

By default, Avalanche-CLI uses the first address of the derivation, and that address needs funds to issue the TXs to create the Subnet and add validators.

To get the first `Mainnet` address of your Ledger device, first make sure it is connected, unblocked, and running the Avalanche app. Then execute the `key list` command:

```
avalanche key list --ledger 0 --mainnet
```

KIND	NAME	CHAIN	ADDRESS	BALANCE	NETWORK
ledger	index 0	P-Chain (Bech32 format)	P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j	11	Mainnet

The command prints the P-Chain address for `Mainnet`, `P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j`, and its balance. You should fund this address with at least 2.5 AVAX to cover TX fees. the TX fee for creating your Subnet costs 2 AVAX. Adding validators costs 0.001 AVAX each. For more details, see [Fees](#)

:::note

You can use the `key list` command to get any Ledger address in the derivation sequence by changing the index parameter from `0` to the one desired, or to a list of them (for example: `2`, or `0,4,7`). Also, you can ask for addresses on `Fuji` with the `--fuji` parameter, and local networks with the `--local` parameter.

:::

Funding the Ledger

A new Ledger device has no funds on the addresses it controls. You'll need to send funds to it by exporting them from the C-Chain to the P-Chain using [Avalanche's Web Wallet](#).

You can load the Ledger's C-Chain address in the web wallet or load in a different private key. You can transfer funds from the C-Chain to the P-Chain by clicking on the Cross Chain on the left side of the web wallet. See this [tutorial](#) for more instructions.

Deploy the Subnet

With your Ledger unlocked and running the Avalanche app, run

```
avalanche subnet deploy testsubnet
```

This is going to start a new prompt series.

```

Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to deploy on:
  Local Network
  Fuji
  ▶ Mainnet

```

This tutorial is about deploying to `Mainnet`, so navigate with the arrow keys to `Mainnet` and hit enter.

```

✓ Mainnet
Deploying [testsubnet] to Mainnet

```

After that, CLI shows the `Mainnet` Ledger address used to fund the deployment:

```
Ledger address: P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j
```

The deployment requires running a [createSubnet transaction](#) and a [createBlockchain transaction](#), and so this first Ledger address must have the funds to issue both operations.

This tutorial creates a permissioned Subnet. As such, you must specify which P-Chain addresses can control the Subnet. These addresses are known as `Control Keys`. The CLI can automatically set your Ledger's address as the sole control key or the user may specify a custom list.

:::warning

In production Subnets, you should always use multiple control keys running in a multisig configuration. This tutorial uses a single control key for illustrative purposes only.

For instructions on controlling your Subnet with a multisig, see the [Multisig Deployment Tutorial](#).

:::

```
Configure which addresses may make changes to the subnet.  
These addresses are known as your control keys. You are going to also  
set how many control keys are required to make a subnet change (the threshold).  
Use the arrow keys to navigate: ↑ ↓ ← →  
? How would you like to set your control keys?:  
▶ Use ledger address  
Custom list
```

For this tutorial, opt to use the first Ledger address, so enter at `Use ledger address`. Only this address is going to be able to add or remove validators, or create blockchains on the Subnet.

```
Your Subnet's control keys: [P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j]  
Your subnet auth keys for chain creation: [P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j]
```

Next, the CLI generates a TX for creating the SubnetID and asks the user to sign it by using the Ledger.

```
*** Please sign subnet creation hash on the ledger device ***
```

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

If the Ledger doesn't have enough funds, the user may see an error message:

```
*** Please sign subnet creation hash on the ledger device ***  
Error: insufficient funds: provided UTXOs need 1000000000 more units of asset  
"U8iRqJoiJm8xZHaacmvYyZVwqQx6uDNtQeP3CQ6fcgQk3JqnK"
```

If successful, the CLI next asks you to sign a chain creation TX.

```
Subnet has been created with ID: 2UUCLbdGqawRDND7kHjq3zXXMPdiycG2bkyuRzYMnuTSDr6db. Now creating blockchain...  
*** Please sign blockchain creation hash on the ledger device ***
```

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

After that, CLI creates the blockchain within the Subnet, and a summary window for the deployment appears:

DEPLOYMENT RESULTS	
Chain Name testsubnet	
Subnet ID 2UUCLbdGqawRDND7kHjq3zXXMPdiycG2bkyuRzYMnuTSDr6db	
VM ID rWlesjm6gy4BtGvxKMpHB2M28MJGFNsqHRY9AmnchdcgeB3ii	
Blockchain ID wNoEemzDER54Zy3iNn66yjUxXmZS9LksZYSUciL89274mHsjG	
RPC URL http://127.0.0.1:9650/ext/bc/wNoEemzDER54Zy3iNn66yjUxXmZS9LksZYSUciL89274mHsjG/rpc	
P-Chain TXID wNoEemzDER54Zy3iNn66yjUxXmZS9LksZYSUciL89274mHsjG	

Well done. You have just created your own Subnet running on `Mainnet`. Now it's time to add your validators.

Add a Validator

:::info

Adding a validator on a Subnet requires that the node is already a validator on the primary network, which means that your node has **fully bootstrapped**.

See [here](#) on how to become a validator.

:::

This new Subnet is cool, but it doesn't have any dedicated validators and can't process transactions yet.

To add a validator to your Subnet, you must add the desired node's NodeID to the Subnet's validator allow list by issuing an `addValidator` transaction and instruct the node to start syncing the Subnet by updating its local configs. You need to repeat this process for every validator you add to the network.

You can run the `addValidator` transactions from the same box that deployed the Subnet.

Submit addValidator TX to Mainnet

Start by running the `addvalidator` command and adding the name of the Subnet. To be clear, this does *not start or run* a validator, it only whitelists the node as a recognized validator on the Subnet.

```
avalanche subnet addValidator testsubnet
```

First choose `Mainnet` as the network to add the Subnet validator to.

```
Use the arrow keys to navigate: ↑ ↓ ← →  
? Choose a network to add validator to.:  
  Fuji  
▶ Mainnet
```

Because this operation issues a new [transaction](#), the CLI needs the control keys to sign the operation. Because this tutorial only uses one control key in the Subnet, the process looks slightly different if you use multiple controls keys. The address needs to pay a TX fee of 0.001 AVAX.

```
Your subnet auth keys for add validator tx creation: [P-avaxlucykh6ls8thqpuwhg3vp8vvu6spg5e8tp8a25j]
```

Set NodeID

Now enter the [NodeID](#) of the validator.

```
Next, we need the NodeID of the validator you want to whitelist.
```

```
Check https://docs.avax.network/apis/avalanchego/apis/info#infogetnodeid for instructions about how  
to query the NodeID from your node  
(Edit host IP address and port to match your deployment, if needed).  
✓ What is the NodeID of the validator you'd like to whitelist?: NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg█
```

Note, this ID is intentionally modified to prevent replication.

Set Stake Weight

Select 30 as the stake weight. You can learn more about the stake weight parameter in [addSubnetValidator](#) under the `weight` section.

:::warning

The stake weight of all your validators should sum to at least 100.

...

```
Use the arrow keys to navigate: ↑ ↓ ← →  
? What stake weight would you like to assign to the validator?:  
  Default (20)  
▶ Custom
```

```
✓ What stake weight would you like to assign to the validator?: 30
```

Set Validation Start Time

Next, specify when the validator is going to start validating. The time must be in the future. You can use the custom option to enter a specific date in `YYYY-MM-DD HH:MM:SS` format. Follow the default this time:

```
Use the arrow keys to navigate: ↑ ↓ ← →  
? Start time:  
  ▶ Start in one minute  
  Custom
```

:::warning

If the `join` command isn't successfully completed before this time elapses and the validator's stake weight is >20% of the Subnet, the Subnet may have down time.

Production Subnets should ensure they have sufficient time to setup their validators **before** validation begins.

...

Set Validation Duration

Finally, specify how long it's going to be validating:

```
✓ Start in one minute
Use the arrow keys to navigate: ↑ ↓ ← →
? How long should your validator validate for?:
▶ Until primary network validator expires
Custom
```

If you choose `Custom` here, you need to enter a **duration**, which is a time span expressed in hours. For example, say `200 days = 24 * 200 = 4800h`

```
✓ How long should this validator be validating? Enter a duration, e.g. 8760h: 4800h
```

The CLI shows the user an actual date:

```
? Your validator is going to finish staking by 2023-06-10 13:07:58:
▶ Yes
No
```

Confirm if correct.

Issue the TX

At this point the prompt series is complete and the CLI attempts the transaction:

```
NodeID: NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Ig
Network: Mainnet
Start time: 2022-11-22 13:07:58
End time: 2023-06-10 13:07:58
Weight: 30
Inputs complete, issuing transaction to add the provided validator information...
```

The CLI shows the Ledger address:

```
Ledger address: P-avaxlucykh6ls8thgpuwhg3vp8vvu6spg5e8tp8a25j
```

At this point, if the Ledger address isn't the control key for the Subnet, the user receives an error:

```
Error: wallet doesn't contain subnet auth keys
exit status 1
```

If the Ledger doesn't have enough funds, the following error message appears:

```
*** Please sign subnet creation hash on the ledger device ***
Error: insufficient funds: provided UTXOs need 1000000 more units of asset "U8iRqJoiJm8xZHaacmvYyZVwqQx6uDNTQeP3CQ6fcgQk3JqnK"
```

Otherwise, both the CLI and the Ledger request to sign the TX:

```
*** Please sign add validator hash on the ledger device ***
```

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

This might take a couple of seconds. After, it prints:

```
Transaction successful, transaction ID: r3tJ4Wr2CWA8AaticmFrKdKgAs5AhW2wwWTaQHRBZKwJhsXzb
```

This means the node is now a validator on the given Subnet on `Mainnet`! However, your work isn't complete. You **must** finish the [Join a Subnet](#) section otherwise your Subnet risks downtime.

You can get the P-Chain TX id information on [Avalanche Explorer](#)

Subnet Export

Because you need to setup multiple validators on multiple different machines, you need to export your Subnet's configuration and import it on each validator.

```
avalanche subnet export testsubnet
✓ Enter file path to write export data to: /tmp/testsubnet-export.dat
```

The file is in text format and you shouldn't change it. You can use it to import the configuration on a different machine.

Subnet Import

To import a VM configuration, move the file you exported in the previous section to your desired machine and issue the `import` command with the path to the file.

```
 avalanche subnet import /tmp/testsubnet-export.dat
Subnet imported successfully
```

After this the whole Subnet configuration should be available on the target machine:

```
 avalanche subnet list
+-----+-----+-----+-----+
| SUBNET | CHAIN | CHAIN ID | TYPE | DEPLOYED |
+-----+-----+-----+-----+
| testsubnet | testsubnet | 3333 | SubnetEVM | No |
+-----+-----+-----+-----+
```

Join a Subnet

To configure your validator to sync a specific Subnet, run the `join` command. This is a bit of a special command. The `join` command is going to either just *print the required instructions* for your already running node or is going to attempt to edit your config files automatically. If you are running the CLI on the same box as your validator, you should run the commands in automated mode. If you don't want to run Avalanche-CLI on the same box as your validator, use the manual commands.

```
 avalanche subnet join testsubnet
```

Select Mainnet

First ask for network for which the validator is joining. Choose `Mainnet`:

```
 Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to validate on (this command only supports public networks):
  Fuji
▶ Mainnet
```

Check Validator List Status

Because you deployed a permissioned Subnet, not just any primary network validator may validate the Subnet. The Subnet's control key holders must have first added the desired NodeID to the Subnet's validator list with the `addValidator` command described in the preceding section. Avalanche-CLI checks that your node is authorized to validate the target Subnet.

```
 Would you like to check if your node is allowed to join this subnet?
If not, the subnet's control key holder must call avalanche subnet
addValidator with your NodeID.
Use the arrow keys to navigate: ↑ ↓ ← →
? Check whitelist?:
  ▶ Yes
  No
```

If you followed the steps in [Add a Validator](#), everything should work as expected.

Setup Node Automatically

There are now two choices possible: automatic and Manual configuration. As mentioned earlier, "Automatic" attempts to edit your config file and sets up your plugin directory, while "Manual" just prints the required config to the screen. To run in automatic mode, you must be running Avalanche-CLI on the same box as the validator node.

Select automatic.

Set Config File

```
 ✓ Automatic
✓ Path to your existing config file (or where it's going to be generated): config.json
```

Provide a path to a config file. If this command runs on the box where your validator is running, then you could point this to the actually used config file, for example `/etc/avalanchego/config.json` - just make sure the tool has **write** access to the file. Or you could just copy the file later. In any case, the tool is going to either try to edit the existing file specified by the given path, or create a new file. Again, set write permissions.

Set Plugin Directory

Next, provide the plugin directory. Each VM runs its own binary, called a plugin. Therefore, you need to copy your VM's plugin binary into AvalancheGo's plugin directory. This directory depends on your AvalancheGo install location.

```
 ✓ Path to your avalanchego plugin dir (likely avalanchego/build/plugins): /home/user/go/src/github.com/ava-
labs/avalanchego/build/plugins
```

The tool doesn't know where exactly it's located so it requires the full path. With the path given, it's going to copy the VM binary to the provided location:

```

✓ Path to your avalanchego plugin dir (likely avalanchego/build/plugins): /home/user/go/src/github.com/avalanchego/avalanchego/build/plugins
VM binary written to /home/user/go/src/github.com/avalanchego/avalanchego/build/plugins/tGBBrMADESojmu5Et9CpbGCrmVf9fiAJtZM5ZJ3YVDj5JTU2qw
This is going to edit your existing config file. This edit is nondestructive,
but it's always good to have a backup.
Use the arrow keys to navigate: ↑ ↓ ← →
? Proceed?:
▶ Yes
No

```

Hitting `Yes` writes the necessary file:

```

✓ Yes
The config file has been edited. To use it, make sure to start the node with the '--config-file' option, e.g.

./build/avalanchego --config-file config.json

(using your binary location). The node has to be restarted for the changes to take effect.

```

Restart the Node

It's required to restart the node.

Setup Node Manually

By choosing "Manual" instead, the tool prints *instructions*. The user is going to have to follow these instructions and apply them to the node. Note that the IDs for the VM and Subnet are different for each Subnet deployment and you shouldn't copy them from this tutorial.

```

✓ Manual

To setup your node, you must do two things:

1. Add your VM binary to your node's plugin directory
2. Update your node config to start validating the subnet

To add the VM to your plugin directory, copy or scp from /tmp/tGBBrMADESojmu5Et9CpbGCrmVf9fiAJtZM5ZJ3YVDj5JTU2qw

If you installed avalanchego manually, your plugin directory is likely
avalanchego/build/plugins.

If you start your node from the command line WITHOUT a config file (e.g. via command
line or systemd script), add the following flag to your node's startup command:

--track-subnets=2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H
(if the node already has a track-subnets config, append the new value by
comma-separating it).

For example:
./build/avalanchego --network-id=Mainnet --track-subnets=2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H

If you start the node via a JSON config file, add this to your config file:
track-subnets: 2b175hLJhGdj3CzgXENso9CmwMgejaCQXhMFzBsm8hXbH2MF7H

TIP: Try this command with the --avalanchego-config flag pointing to your config file,
this tool is going to try to update the file automatically (make sure it can write to it).

After you update your config, you are going to need to restart your node for the changes to
take effect.

```

Going Live

Once all of your validators have joined the network, you are ready to issue transactions to your Subnet.

For the safety of your validators, you should setup dedicated API nodes to process transactions, but for test purposes, you can issue transactions directly to one of your validator's RPC interface.

How to Build a Simple Rust VM

This is part of a series of tutorials for building a Virtual Machine (VM):

- [Introduction to VMs](#)
- [How to Build a Simple Golang VM](#)
- [How to Build a Complex Golang VM](#)

- How to Build a Simple Rust VM (this article)

Introduction

The Avalanche Rust SDK is a developer toolkit composed of powerful building blocks and primitive types. This tutorial will walk you through the creation of a simple VM known as the [TimestampVM RS](#) using the Rust SDK. Each block in the TimestampVM's blockchain contains a monotonically increasing timestamp when the block was created and a 32-byte payload of data.

Prerequisites

- Install the latest stable version of Rust using [rustup](#).
- Bookmark and review the [avalanche-types](#) GitHub repository specifically the traits and helpers defined in the [subnet/rpc](#) mod.
- For developers new to Rust please visit the free online book [The Rust Programming Language](#).

If you have experimented with our Golang example VMs you will find the Rust SDK fairly familiar. Completely new to creating a custom VM on Avalanche? No problem please review [Introduction to VMs](#).

Components

A VM defines how a blockchain should be built. A block is populated with a transaction which mutates the state of the blockchain when executed. By executing a series of blocks chronologically, anyone can verify and reconstruct the state of the blockchain at an arbitrary point in time.

The TimestampVM RS repository has a few components to handle the lifecycle of tasks from a transaction being issued to a block being accepted across the network:

- **Mempool** - Stores pending transactions that haven't been finalized yet.
- **Block** - Defines the block format, how to verify it, and how it should be accepted or rejected across the network
- **Virtual Machine** - Application-level logic. Implements the VM trait needed to interact with Avalanche consensus and defines the blueprint for the blockchain.
- **Service** - Exposes APIs so users can interact with the VM.
- **State** - Manages both in memory and persistent states.

TimestampVM RS Implementation

The TimestampVM RS implements the [snowman::block::ChainVM](#) trait. Below you will find additional documentation on the trait methods. To assist with a logical understanding of the expectations for these methods please see the code examples below.

Additional Documentation

- [ChainVM GoDoc](#)
- [Avalanche Proto Docs](#)
- [Snowman VMs](#)

Now we know the traits (interfaces) our VM must implement and the libraries we can use to build a VM using the Rust SDK.

Let's write our VM, which implements `snowman::block::ChainVM` and whose blocks implement `snowman::Block`. You can also follow the code in the [TimestampVM RS repository](#).

State

`State` manages block and chain states for this VM, both in-memory and persistent.

```
#[derive(Clone)]
pub struct State {
    pub db: Arc<RwLock<Box<dyn subnet::rpc::database::Database + Send + Sync>>,
    // Maps block Id to Block.
    // Each element is verified but not yet accepted/rejected (e.g., preferred).
    pub verified_blocks: Arc<RwLock<HashMap<ids::Id, Block>>,
}

impl Default for State {
    fn default() -> State {
        Self {
            db: Arc::new(RwLock::new(subnet::rpc::database::memdb::Database::new())),
            verified_blocks: Arc::new(RwLock::new(HashMap::new())),
        }
    }
}

const LAST_ACCEPTED_BLOCK_KEY: &[u8] = b"last_accepted_block";

const STATUS_PREFIX: u8 = 0x0;

const DELIMITER: u8 = b'';

/// Returns a vec of bytes used as a key for identifying blocks in state.
/// 'STATUS_PREFIX' + 'BYTE_DELIMITER' + [block_id]
fn block_with_status_key(blk_id: &ids::Id) -> Vec<u8> {
    let mut k: Vec<u8> = Vec::with_capacity(ids::LEN + 2);
    k.push(STATUS_PREFIX);
    k.push(DELIMITER);
    k.push(*blk_id);
    return k;
}
```

```

    k.push(STATUS_PREFIX);
    k.push(DELIMITER);
    k.extend_from_slice(&blk_id.to_vec());
    k
}

/// Wraps a [`Block`] (crate::block::Block) and its status.
/// This is the data format that [`State`] (State) uses to persist blocks.
#[derive(Serialize, Deserialize, Clone)]
struct BlockWithStatus {
    block_bytes: Vec<u8>,
    status: choices::status::Status,
}

impl BlockWithStatus {
    fn encode(&self) -> io::Result<Vec<u8>> {
        serde_json::to_vec(&self).map_err(|e| {
            Error::new(
                ErrorKind::Other,
                format!("failed to serialize BlockStatus to JSON bytes: {}", e),
            )
        })
    }
}

fn from_slice(d: impl AsRef<[u8]>) -> io::Result<Self> {
    let dd = d.as_ref();
    serde_json::from_slice(dd).map_err(|e| {
        Error::new(
            ErrorKind::Other,
            format!("failed to deserialize BlockStatus from JSON: {}", e),
        )
    })
}

impl State {
    /// Persists the last accepted block Id.
    pub async fn set_last_accepted_block(&self, blk_id: &ids::Id) -> io::Result<()> {
        let mut db = self.db.write().await;
        db.put(LAST_ACCEPTED_BLOCK_KEY, &blk_id.to_vec())
            .await
        .map_err(|e| {
            Error::new(
                ErrorKind::Other,
                format!("failed to put last accepted block: {:?}", e),
            )
        })
    }

    /// Returns "true" if there's a last accepted block found.
    pub async fn has_last_accepted_block(&self) -> io::Result<bool> {
        let db = self.db.read().await;
        match db.get(LAST_ACCEPTED_BLOCK_KEY).await {
            Ok(found) => Ok(found),
            Err(e) => Err(Error::new(
                ErrorKind::Other,
                format!("failed to load last accepted block: {}", e),
            )),
        }
    }

    /// Returns the last accepted block Id.
    pub async fn get_last_accepted_block_id(&self) -> io::Result<ids::Id> {
        let db = self.db.read().await;
        match db.get(LAST_ACCEPTED_BLOCK_KEY).await {
            Ok(d) => Ok(ids::Id::from_slice(&d)),
            Err(e) => {
                if subnet::rpc::database::errors::is_not_found(&e) {
                    return Ok(ids::Id::empty());
                }
                Err(e)
            }
        }
    }

    /// Adds a block to "verified_blocks".
    pub async fn add_verified(&mut self, block: &Block) {
        let blk_id = block.id();

```

```

log::info!("verified added {blk_id}");

let mut verified_blocks = self.verified_blocks.write().await;
verified_blocks.insert(blk_id, block.clone());
}

/// Removes a block from "verified_blocks".
pub async fn remove_verified(&mut self, blk_id: &ids::Id) {
    let mut verified_blocks = self.verified_blocks.write().await;
    verified_blocks.remove(blk_id);
}

/// Returns "true" if the block Id has been already verified.
pub async fn has_verified(&self, blk_id: &ids::Id) -> bool {
    let verified_blocks = self.verified_blocks.read().await;
    verified_blocks.contains_key(blk_id)
}

/// Writes a block to the state storage.
pub async fn write_block(&mut self, block: &Block) -> io::Result<()> {
    let blk_id = block.id();
    let blk_bytes = block.to_slice()?;

    let mut db = self.db.write().await;

    let blk_status = BlockWithStatus {
        block_bytes: blk_bytes,
        status: block.status(),
    };
    let blk_status_bytes = blk_status.encode()?;

    db.put(&block_with_status_key(&blk_id), &blk_status_bytes)
        .await
        .map_err(|e| Error::new(ErrorKind::Other, format!("failed to put block: {:?}", e)))
}

/// Reads a block from the state storage using the block_with_status_key.
pub async fn get_block(&self, blk_id: &ids::Id) -> io::Result<Block> {
    // check if the block exists in memory as previously verified.
    let verified_blocks = self.verified_blocks.read().await;
    if let Some(b) = verified_blocks.get(blk_id) {
        return Ok(b.clone());
    }

    let db = self.db.read().await;

    let blk_status_bytes = db.get(&block_with_status_key(blk_id)).await?;
    let blk_status = BlockWithStatus::from_slice(&blk_status_bytes)?;

    let mut blk = Block::from_slice(&blk_status.block_bytes)?;
    blk.set_status(blk_status.status);

    Ok(blk)
}
}

```

Block

This implementation of `snowman::Block` provides the VM with storage, retrieval and status of blocks.

Block is a block on the chain. Each block contains:

- ParentID
- Height
- Timestamp
- A piece of data (hex encoded string)

```

#[serde_as]
#[derive(Serialize, Deserialize, Clone, Derivative)]
#[derivative(Debug, PartialEq, Eq)]
pub struct Block {
    /// The block Id of the parent block.
    parent_id: ids::Id,
    /// This block's height.
    /// The height of the genesis block is 0.
    height: u64,
    /// Unix second when this block was proposed.
    timestamp: u64,
}

```

```

/// Arbitrary data.
#[serde(as = "Hex0xBytes")]
data: Vec<u8>,

/// Current block status.
#[serde(skip)]
status: choices::status::Status,
/// This block's encoded bytes.
#[serde(skip)]
bytes: Vec<u8>,
/// Generated block Id.
#[serde(skip)]
id: ids::Id,

/// Reference to the Vm state manager for blocks.
#[derivative(Debug = "ignore", PartialEq = "ignore")]
#[serde(skip)]
state: state::State,
}

impl Default for Block {
    fn default() -> Self {
        Self::default()
    }
}

impl Block {
    pub fn default() -> Self {
        Self {
            parent_id: ids::Id::empty(),
            height: 0,
            timestamp: 0,
            data: Vec::new(),
            status: choices::status::Status::default(),
            bytes: Vec::new(),
            id: ids::Id::empty(),
            state: state::State::default(),
        }
    }

    pub fn new(
        parent_id: ids::Id,
        height: u64,
        timestamp: u64,
        data: Vec<u8>,
        status: choices::status::Status,
    ) -> io::Result<Self> {
        let mut b = Self {
            parent_id,
            height,
            timestamp,
            data,
            ..Default::default()
        };

        b.status = status;
        b.bytes = b.to_slice()?;
        b.id = ids::Id::sha256(&b.bytes);

        Ok(b)
    }
}

pub fn to_json_string(&self) -> io::Result<String> {
    serde_json::to_string(&self).map_err(|e| {
        Error::new(
            ErrorKind::Other,
            format!("failed to serialize Block to JSON string {}", e),
        )
    })
}

/// Encodes the ['Block'](Block) to JSON in bytes.
pub fn to_slice(&self) -> io::Result<Vec<u8>> {
    serde_json::to_vec(&self).map_err(|e| {
        Error::new(
            ErrorKind::Other,

```

```

        format!("failed to serialize Block to JSON bytes {}", e),
    )
}
}

/// Loads [`Block`](Block) from JSON bytes.
pub fn from_slice(d: impl AsRef<[u8]>) -> io::Result<Self> {
    let dd = d.as_ref();
    let mut b: Self = serde_json::from_slice(dd).map_err(|e| {
        Error::new(
            ErrorKind::Other,
            format!("failed to deserialize Block from JSON {}", e),
        )
    })?;

    b.bytes = dd.to_vec();
    b.id = ids::Id::sha256(&b.bytes);

    Ok(b)
}

/// Returns the parent block Id.
pub fn parent_id(&self) -> ids::Id {
    self.parent_id
}

/// Returns the height of this block.
pub fn height(&self) -> u64 {
    self.height
}

/// Returns the timestamp of this block.
pub fn timestamp(&self) -> u64 {
    self.timestamp
}

/// Returns the data of this block.
pub fn data(&self) -> &[u8] {
    &self.data
}

/// Returns the status of this block.
pub fn status(&self) -> choices::status::Status {
    self.status.clone()
}

/// Updates the status of this block.
pub fn set_status(&mut self, status: choices::status::Status) {
    self.status = status;
}

/// Returns the byte representation of this block.
pub fn bytes(&self) -> &[u8] {
    &self.bytes
}

/// Returns the ID of this block
pub fn id(&self) -> ids::Id {
    self.id
}

/// Updates the state of the block.
pub fn set_state(&mut self, state: state::State) {
    self.state = state;
}

/// Verifies [`Block`](Block) properties (e.g., heights),
/// and once verified, records it to the [`State`](crate::state::State).
pub async fn verify(&mut self) -> io::Result<()> {
    if self.height == 0 && self.parent_id == ids::Id::empty() {
        log::debug!(
            "block {} has an empty parent Id since it's a genesis block -- skipping verify",
            self.id
        );
        self.state.add_verified(&self.clone()).await;
        return Ok(());
    }
}

```

```

// if already exists in database, it means it's already accepted
// thus no need to verify once more
if self.state.get_block(&self.id).await.is_ok() {
    log::debug!("block {} already verified", self.id);
    return Ok(());
}

let prnt_blk = self.state.get_block(&self.parent_id).await?;

// ensure the height of the block is immediately following its parent
if prnt_blk.height != self.height - 1 {
    return Err(Error::new(
        ErrorKind::InvalidData,
        format!(
            "parent block height {} != current block height {} - 1",
            prnt_blk.height, self.height
        ),
    )));
}

// ensure block timestamp is after its parent
if prnt_blk.timestamp > self.timestamp {
    return Err(Error::new(
        ErrorKind::InvalidData,
        format!(
            "parent block timestamp {} > current block timestamp {}",
            prnt_blk.timestamp, self.timestamp
        ),
    )));
}

// ensure block timestamp is no more than an hour ahead of this nodes time
if self.timestamp >= (Utc::now() + Duration::hours(1)).timestamp() as u64 {
    return Err(Error::new(
        ErrorKind::InvalidData,
        format!(
            "block timestamp {} is more than 1 hour ahead of local time",
            self.timestamp
        ),
    )));
}

// add newly verified block to memory
self.state.add_verified(&self.clone()).await;
Ok(())
}

/// Mark this ['Block'](Block) accepted and updates ['State'](crate::state::State) accordingly.
pub async fn accept(&mut self) -> io::Result<()> {
    self.set_status(choices::status::Status::Accepted);

    // only decided blocks are persistent -- no reorg
    self.state.write_block(&self.clone()).await?;
    self.state.set_last_accepted_block(&self.id()).await?;

    self.state.remove_verified(&self.id()).await;
    Ok(())
}

/// Mark this ['Block'](Block) rejected and updates ['State'](crate::state::State) accordingly.
pub async fn reject(&mut self) -> io::Result<()> {
    self.set_status(choices::status::Status::Rejected);

    // only decided blocks are persistent -- no reorg
    self.state.write_block(&self.clone()).await?;

    self.state.remove_verified(&self.id()).await;
    Ok(())
}
}

```

verify

This method verifies that a block is valid and stores it in the memory. It is important to store the verified block in the memory and return them in the `vm.get_block()` method.

```

pub async fn verify(&mut self) -> io::Result<()> {
    if self.height == 0 && self.parent_id == ids::Id::empty() {
        log::debug!(
            "block {} has an empty parent Id since it's a genesis block -- skipping verify",
            self.id
        );
        self.state.add_verified(&self.clone()).await;
        return Ok(());
    }

    // if already exists in database, it means it's already accepted
    // thus no need to verify once more
    if self.state.get_block(&self.id).await.is_ok() {
        log::debug!("block {} already verified", self.id);
        return Ok(());
    }

    let prnt_blk = self.state.get_block(&self.parent_id).await?;

    // ensure the height of the block is immediately following its parent
    if prnt_blk.height != self.height - 1 {
        return Err(Error::new(
            ErrorKind::InvalidData,
            format!(
                "parent block height {} != current block height {} - 1",
                prnt_blk.height, self.height
            ),
        )));
    }

    // ensure block timestamp is after its parent
    if prnt_blk.timestamp > self.timestamp {
        return Err(Error::new(
            ErrorKind::InvalidData,
            format!(
                "parent block timestamp {} > current block timestamp {}",
                prnt_blk.timestamp, self.timestamp
            ),
        )));
    }

    // ensure block timestamp is no more than an hour ahead of this nodes time
    if self.timestamp >= (Utc::now() + Duration::hours(1)).timestamp() as u64 {
        return Err(Error::new(
            ErrorKind::InvalidData,
            format!(
                "block timestamp {} is more than 1 hour ahead of local time",
                self.timestamp
            ),
        ));
    }

    // add newly verified block to memory
    self.state.add_verified(&self.clone()).await;
    Ok(())
}

```

accept

Called by the consensus engine to indicate this block is accepted.

```

pub async fn accept(&mut self) -> io::Result<()> {
    self.set_status(choices::status::Status::Accepted);

    // only decided blocks are persistent -- no reorg
    self.state.write_block(&self.clone()).await?;
    self.state.set_last_accepted_block(&self.id()).await?;

    self.state.remove_verified(&self.id()).await;
    Ok(())
}

```

reject

Called by the consensus engine to indicate the block is rejected.

```

pub async fn reject(&mut self) -> io::Result<()> {
    self.set_status(choices::status::Status::Rejected);

    // only decided blocks are persistent -- no reorg
    self.state.write_block(&self.clone()).await?;

    self.state.remove_verified(&self.id()).await;
    Ok(())
}

```

Block Field Methods

These methods are required by the `snowman::Block` trait.

```

impl subnet::rpc::consensus::snowman::Block for Block {
    async fn bytes(&self) -> &[u8] {
        return self.bytes.as_ref();
    }

    async fn to_bytes(&self) -> io::Result<Vec<u8>> {
        self.to_slice()
    }

    async fn height(&self) -> u64 {
        self.height
    }

    async fn timestamp(&self) -> u64 {
        self.timestamp
    }

    async fn parent(&self) -> ids::Id {
        self.parent_id.clone()
    }

    async fn verify(&mut self) -> io::Result<()> {
        self.verify().await
    }
}

```

Helper Functions

These methods are convenience methods for blocks.

init

Initializes a block from a bytes slice and status.

```

impl subnet::rpc::consensus::snowman::Initializer for Block {
    async fn init(&mut self, bytes: &[u8], status: choices::status::Status) -> io::Result<()> {
        *self = Block::from_slice(bytes)?;
        self.status = status;

        Ok(())
    }
}

```

set_status

Updates the status of this block.

```

impl subnet::rpc::consensus::snowman::StatusWriter for Block {
    async fn set_status(&mut self, status: choices::status::Status) {
        self.set_status(status)
    }
}

```

Coding the Virtual Machine

Now, let's look at our timestamp VM implementation, which implements the `block::ChainVM` trait.

```

pub struct Vm {
    /// Maintains the Vm-specific states.
    pub state: Arc<RwLock<VmState>>,
    pub app_sender: Option<Box<dyn subnet::rpc::common::appsender::AppSender + Send + Sync>>,
}

```

```

/// A queue of data that have not been put into a block and proposed yet.
/// Mempool is not persistent, so just keep in memory via Vm.
pub mempool: Arc<RwLock<VecDeque<Vec<u8>>>,
}

/// Represents VM-specific states.
/// Defined in a separate struct, for interior mutability in [Vm](Vm).
/// To be protected with `Arc` and `RwLock`.
pub struct VmState {
    pub ctx: Option<subnet::rpc::context::Context>,
    pub version: Version,
    pub genesis: Genesis,

    /// Represents persistent Vm state.
    pub state: Option<state::State>,
    /// Currently preferred block Id.
    pub preferred: ids::Id,
    /// Channel to send messages to the snowman consensus engine.
    pub to_engine: Option<Sender<subnet::rpc::common::Message>>,
    /// Set "true" to indicate that the Vm has finished bootstrapping
    /// for the chain.
    pub bootstrapped: bool,
}

```

initialize

This method is called when a new instance of VM is initialized. Genesis block is created under this method.

```

async fn initialize(
    &mut self,
    ctx: Option<subnet::rpc::context::Context>,
    db_manager: Box<dyn subnet::rpc::database::manager::Manager + Send + Sync>,
    genesis_bytes: &[u8],
    _upgrade_bytes: &[u8],
    _config_bytes: &[u8],
    to_engine: Sender<subnet::rpc::common::Message>,
    _fxs: &[subnet::rpc::common::Vm::Fx],
    app_sender: Box<dyn subnet::rpc::common::appsender::AppSender + Send + Sync>,
) -> io::Result<()> {
    log::info!("initializing Vm");
    let mut vm_state = self.state.write().await;

    vm_state.ctx = ctx;

    let version =
        Version::parse(VERSION).map_err(|e| Error::new(ErrorKind::Other, e.to_string()))?;
    vm_state.version = version;

    let genesis = Genesis::from_slice(genesis_bytes)?;
    vm_state.genesis = genesis;

    let current = db_manager.current().await?;
    let state = state::State {
        db: Arc::new(RwLock::new(current.db)),
        verified_blocks: Arc::new(RwLock::new(HashMap::new())),
    };
    vm_state.state = Some(state.clone());

    vm_state.to_engine = Some(to_engine);

    self.app_sender = Some(app_sender);

    let has_last_accepted = state.has_last_accepted_block().await?;
    if has_last_accepted {
        let last_accepted_blk_id = state.get_last_accepted_block_id().await?;
        vm_state.preferred = last_accepted_blk_id;
        log::info!("Initialized Vm with last accepted block {last_accepted_blk_id}");
    } else {
        let mut genesis_block = Block::new(
            ids::Id::empty(),
            0,
            0,
            vm_state.genesis.data.as_bytes().to_vec(),
            choices::status::Status::default(),
        )?;
        genesis_block.set_state(state.clone());
        genesis_block.accept().await?;
    }
}

```

```

        let genesis_blk_id = genesis_block.id();
        vm_state.preferred = genesis_blk_id;
        log::info!("initialized Vm with genesis block {genesis_blk_id}");
    }

    self.mempool = Arc::new(RwLock::new(VecDeque::with_capacity(100)));

    log::info!("successfully initialized Vm");
    Ok(())
}

```

create_handlers

Registers handlers defined in `api::chain_handlers::Service`. See [below](#) for more on APIs.

```

/// Creates VM-specific handlers.
async fn create_handlers(
    &mut self,
) -> io::Result<HashMap<String, subnet::rpc::common::http_handler::HttpHandler>> {
    let svc = api::chain_handlers::Service::new(self.clone());
    let mut handler = jsonrpc_core::IoHandler::new();
    handler.extend_with(api::chain_handlers::Rpc::to_delegate(svc));

    let http_handler = subnet::rpc::common::http_handler::HttpHandler::new_from_u8(0, handler)
        .map_err(|_| Error::from(ErrorKind::InvalidData))?;

    let mut handlers = HashMap::new();
    handlers.insert("/rpc".to_string(), http_handler);
    Ok(handlers)
}

```

create_static_handlers

Registers handlers defined in `api::chain_handlers::Service`. See [below](#) for more on APIs.

```

async fn create_static_handlers(
    &mut self,
) -> io::Result<HashMap<String, subnet::rpc::common::http_handler::HttpHandler>> {
    let svc = api::static_handlers::Service::new(self.clone());
    let mut handler = jsonrpc_core::IoHandler::new();
    handler.extend_with(api::static_handlers::Rpc::to_delegate(svc));

    let http_handler = subnet::rpc::common::http_handler::HttpHandler::new_from_u8(0, handler)
        .map_err(|_| Error::from(ErrorKind::InvalidData))?;

    let mut handlers = HashMap::new();
    handlers.insert("/static".to_string(), http_handler);
    Ok(handlers)
}

```

build_block

Builds a new block from mempool data and returns it. This is primarily requested by the consensus engine.

```

async fn build_block(
    &self,
) -> io::Result<Box<dyn subnet::rpc::consensus::snowman::Block + Send + Sync>> {
    let mut mempool = self.mempool.write().await;

    log::info!("build_block called for {} mempool", mempool.len());
    if mempool.is_empty() {
        return Err(Error::new(ErrorKind::Other, "no pending block"));
    }

    let vm_state = self.state.read().await;
    if let Some(state) = &vm_state.state {
        self.notify_block_ready().await;
    }

    // "state" must have preferred block in cache/verified_block
    // otherwise, not found error from rpcchainvm database
    let prnt_blk = state.get_block(&vm_state.preferred).await?;
    let unix_now = Utc::now().timestamp() as u64;

    let first = mempool.pop_front().unwrap();
    let mut block = Block::new(
        prnt_blk.id(),
        prnt_blk.height() + 1,
    );

```

```

        unix_now,
        first,
        choices::status::Status::Processing,
    )?;
    block.set_state(state.clone());
    block.verify().await?;

    log::info!("successfully built block");
    return Ok(Box::new(block));
}

Err(Error::new(ErrorKind::NotFound, "state manager not found"))
}

```

`notify_block_ready`

Signals the consensus engine that a new block is ready to be created. After this is sent the consensus engine will call back to `vm.build_block`.

```

pub async fn notify_block_ready(&self) {
    let vm_state = self.state.read().await;
    if let Some(engine) = &vm_state.to_engine {
        engine
            .send(subnet::rpc::common::message::Message::PendingTxs)
            .await
            .unwrap_or_else(|e| log::warn!("dropping message to consensus engine: {}", e));

        log::info!("notified block ready!");
    } else {
        log::error!("consensus engine channel failed to initialized");
    }
}

```

`get_block`

Returns the block with the given block ID.

```

impl subnet::rpc::snowman::block::Getter for Vm {
    async fn get_block(
        &self,
        blk_id: ids::Id,
    ) -> io::Result<Box<dyn subnet::rpc::consensus::snowman::Block + Send + Sync>> {
        let vm_state = self.state.read().await;
        if let Some(state) = &vm_state.state {
            let block = state.get_block(&blk_id).await?;
            return Ok(Box::new(block));
        }

        Err(Error::new(ErrorKind::NotFound, "state manager not found"))
    }
}

```

`propose_block`

Proposes arbitrary data to mempool and notifies that a block is ready for builds. Other VMs may optimize mempool with more complicated batching mechanisms.

```

pub async fn propose_block(&self, d: Vec<u8>) -> io::Result<()> {
    let size = d.len();
    log::info!("received propose_block of {} bytes", size);

    if size > PROPOSE_LIMIT_BYTES {
        log::info!("limit exceeded... returning an error...");
        return Err(Error::new(
            ErrorKind::InvalidInput,
            format!(
                "data {}-byte exceeds the limit {}-byte",
                size, PROPOSE_LIMIT_BYTES
            ),
        ));
    }

    let mut mempool = self.mempool.write().await;
    mempool.push_back(d);
    log::info!("proposed {} bytes of data for a block");

    self.notify_block_ready().await;
}

```

```
    Ok(())
}
```

parse_block

Parses a block from its byte representation.

```
impl subnet::rpc::snowman::block::Parser for Vm {
    async fn parse_block(
        &self,
        bytes: &[u8],
    ) -> io::Result<Box<dyn subnet::rpc::consensus::snowman::Block + Send + Sync>> {
        let vm_state = self.state.read().await;
        if let Some(state) = &vm_state.state {
            let mut new_block = Block::from_slice(bytes)?;
            new_block.set_status(choices::status::Status::Processing);
            new_block.set_state(state.clone());
            log::debug!("parsed block {}", new_block.id());

            match state.get_block(&new_block.id()).await {
                Ok(prev) => {
                    log::debug!("returning previously parsed block {}", prev.id());
                    return Ok(Box::new(prev));
                }
                Err(_) => return Ok(Box::new(new_block)),
            };
        }

        Err(Error::new(ErrorKind::NotFound, "state manager not found"))
    }
}
```

set_preference

Sets the container preference of the VM.

```
pub async fn set_preference(&self, id: Ids::Id) -> io::Result<()> {
    let mut vm_state = self.state.write().await;
    vm_state.preferred = id;

    Ok(())
}
```

Mempool

The mempool is a buffer of volatile memory that stores pending transactions. Transactions are stored in the mempool whenever a node learns about a new transaction.

The mempool implementation for `timestampvm-rs` is very simple.

```
mempool: Arc::new(RwLock::new(VecDeque::with_capacity(100))),
```

By using [VecDeque](#) we can have better control of element ordering (ex. `pop_back()`, `pop_front()`).

Static API

:::note

If this method is called, no other method will be called on this VM. Each registered VM will have a single instance created to handle static APIs. This instance will be handled separately from instances created to service an instance of a chain.

:::

```
#[rpc]
pub trait Rpc {
    #[rpc(name = "ping", alias("timestampvm.ping"))]
    fn ping(&self) -> BoxFuture<Result<crate::api::PingResponse>>;
}

/// Implements API services for the static handlers.
pub struct Service {
    pub vm: Vm,
}

impl Service {
    pub fn new(vm: Vm) -> Self {
        Self { vm }
    }
}
```

```

    }

impl Rpc for Service {
    fn ping(&self) -> BoxFuture<Result<crate::api::PingResponse>> {
        log::debug!("ping called");
        Box::pin(async move { Ok(crate::api::PingResponse { success: true }) })
    }
}

```

API

Defines RPCs specific to the chain.

```

#[rpc]
pub trait Rpc {
    /// Pings the VM.
    #[rpc(name = "ping", alias("timestampvm.ping"))]
    fn ping(&self) -> BoxFuture<Result<crate::api::PingResponse>>;

    /// Proposes the arbitrary data.
    #[rpc(name = "proposeBlock", alias("timestampvm.proposeBlock"))]
    fn propose_block(&self, args: ProposeBlockArgs) -> BoxFuture<Result<ProposeBlockResponse>>;

    /// Fetches the last accepted block.
    #[rpc(name = "lastAccepted", alias("timestampvm.lastAccepted"))]
    fn last_accepted(&self) -> BoxFuture<Result<LastAcceptedResponse>>;

    /// Fetches the block.
    #[rpc(name = "getBlock", alias("timestampvm.getBlock"))]
    fn get_block(&self, args: GetBlockArgs) -> BoxFuture<Result<GetBlockResponse>>;
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct ProposeBlockArgs {
    #[serde(with = "avalanche_types::codec::serde::base64_bytes")]
    pub data: Vec<u8>,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct ProposeBlockResponse {
    pub success: bool,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct LastAcceptedResponse {
    pub id: ids::Id,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct GetBlockArgs {
    pub id: String,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct GetBlockResponse {
    pub block: Block,
}

/// Implements API services for the chain-specific handlers.
pub struct Service {
    pub vm: vm::Vm,
}

impl Service {
    pub fn new(vm: vm::Vm) -> Self {
        Self { vm }
    }
}

impl Rpc for Service {
    fn ping(&self) -> BoxFuture<Result<crate::api::PingResponse>> {
        log::debug!("ping called");
        Box::pin(async move { Ok(crate::api::PingResponse { success: true }) })
    }

    fn propose_block(&self, args: ProposeBlockArgs) -> BoxFuture<Result<ProposeBlockResponse>> {
        log::debug!("propose_block called");
    }
}

```

```

let vm = self.vm.clone();

Box::pin(async move {
    vm.propose_block(args.data)
        .await
        .map_err(create_jsonrpc_error)?;
    Ok(ProposeBlockResponse { success: true })
})}

fn last_accepted(&self) -> BoxFuture<Result<LastAcceptedResponse>> {
    log::debug!("last accepted method called");
    let vm = self.vm.clone();

    Box::pin(async move {
        let vm_state = vm.state.read().await;
        if let Some(state) = &vm_state.state {
            let last_accepted = state
                .get_last_accepted_block_id()
                .await
                .map_err(create_jsonrpc_error)?;

            return Ok(LastAcceptedResponse { id: last_accepted });
        }

        Err(Error {
            code: ErrorCode::InternalError,
            message: String::from("no state manager found"),
            data: None,
        })
    })
}

fn get_block(&self, args: GetBlockArgs) -> BoxFuture<Result<GetBlockResponse>> {
    let blk_id = args.id.unwrap();
    log::info!("get_block called for {}", blk_id);

    let vm = self.vm.clone();

    Box::pin(async move {
        let vm_state = vm.state.read().await;
        if let Some(state) = &vm_state.state {
            let block = state
                .get_block(&blk_id)
                .await
                .map_err(create_jsonrpc_error)?;

            return Ok(GetBlockResponse { block });
        }

        Err(Error {
            code: ErrorCode::InternalError,
            message: String::from("no state manager found"),
            data: None,
        })
    })
}
}

```

Below are examples of API calls, in which "2wb1UXxAstB8ywwv4rU2rFCjLgXnhT44hbLPbwpQoGvFb2wRR7" is the blockchain ID.

`timestampvm.getBlock`

Given a valid block ID, returns a serialized block.

```

curl -X POST --data '{
    "jsonrpc": "2.0",
    "id" : 1,
    "method" : "timestampvm.getBlock",
    "params" : [{"id":"SDFFUzkdzWzbJ6YMyssPPNEF5dWLp9q35mEMaLa8Ha2w9aMKoC"}]
}' -H 'content-type:application/json' 127.0.0.1:9650/ext/bc/2wb1UXxAstB8ywwv4rU2rFCjLgXnhT44hbLPbwpQoGvFb2wRR7/rpc

# example response
# {"jsonrpc":"2.0","result":{"block":
{"data":"0x32596655705939524358","height":0,"parent_id":"11111111111111111111111111111111LpoYY","timestamp":0}),"id":1}

```

`timestampvm.proposeBlock`

Proposes arbitrary data for a new block to consensus.

```
# to propose data
echo 1 | base64 | tr -d \\n
# MQo=

curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" : "timestampvm.proposeBlock",
  "params" : [{"data":"MQo="}]
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/2wb1UXxAstB8ywwv4rU2rFCjLgXnhT44hbLPbwpQoGvFb2wRR7/rpc

# example response
# {"jsonrpc":"2.0","result":{"success":true},"id":1}
```

timestampvm.lastAccepted

Returns the ID of the last accepted block.

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "id" : 1,
  "method" : "timestampvm.lastAccepted",
  "params" : []
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/2wb1UXxAstB8ywwv4rU2rFCjLgXnhT44hbLPbwpQoGvFb2wRR7/rpc

# example response
# {"jsonrpc":"2.0","result":{"id":"SDfFUzkdzWZbJ6YMySPPNEF5dWLp9q35mEMaLa8Ha2w9aMKoC"},"id":1}
```

Plugin

In order to make this VM compatible with `go-plugin`, we need to define a `main` package and method, which serves our VM over gRPC so that AvalancheGo can call its methods.

`main.rs`'s contents are:

```
async fn main() -> io::Result<()> {
    let matches = Command::new(APP_NAME)
        .version(crate_version!())
        .about("Timestamp Vm")
        .subcommands(vec![genesis::command(), vm_id::command()])
        .get_matches();

    // ref. https://github.com/env-logger-rs/env_logger/issues/47
    env_logger::init_from_env(
        env_logger::Env::default().filter_or(env_logger::DEFAULT_FILTER_ENV, "info"),
    );

    match matches.subcommand() {
        Some((genesis::NAME, sub_matches)) => {
            let data = sub_matches.get_one:::<String>("DATA").expect("required");
            let genesis = timestampvm::genesis { data: data.clone() };
            println!("{}({genesis})");

            Ok(())
        }

        Some((vm_id::NAME, sub_matches)) => {
            let vm_name = sub_matches.get_one:::<String>("VM_NAME").expect("required");
            let id = subnet::vm_name_to_id(vm_name)?;
            println!("{}({id})");

            Ok(())
        }

        _ => {
            log::info!("starting timestampvm");

            let (stop_ch_tx, stop_ch_rx): (Sender<()>, Receiver<()>) = broadcast::channel(1);
            let vm_server = subnet::rpc::vm::server::Server::new(vm::Vm::new(), stop_ch_tx);
            subnet::rpc::plugin::serve(vm_server, stop_ch_rx).await
        }
    }
}
```

Installing a VM

AvalancheGo searches for and registers VM plugins under the `plugins` [directory](#).

To install the virtual machine onto your node, you need to move the built virtual machine binary under this directory. Virtual machine executable names must be either a full virtual machine ID (encoded in CB58), or a VM alias.

Copy the binary into the `plugins` directory.

```
cp -n <path to your binary> ${GOPATH}/src/github.com/ava-labs/avalanche/go/build/plugins/
```

Node Is Not Running

If your node isn't running yet, you can install all virtual machines under your `plugin` directory by starting the node.

Node Is Already Running

Load the binary with the `loadVMs` API.

```
curl -sX POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.loadVMs",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Confirm the response of `loadVMs` contains the newly installed virtual machine `tGas3T58KzdjCJ32c6GpePhtqo9rrHJ1oR9wFBtCcMgaosthX`. You'll see this virtual machine as well as any others that weren't already installed previously in the response.

```
{
  "jsonrpc": "2.0",
  "result": {
    "newVMs": [
      "tGas3T58KzdjCJ32c6GpePhtqo9rrHJ1oR9wFBtCcMgaosthX": [
        "timestampvm-rs",
        "timestamp-rs"
      ],
      "spdxUxVJQbX85MGxMHbKwlshxMnSqJ3QBzDyDYEh6TLuxqQ": []
    ]
  },
  "id": 1
}
```

Now, this VM's static API can be accessed at endpoints `/ext/vm/timestampvm-rs` and `/ext/vm/timestamp-rs`. For more details about VM configs, see [here](#).

In this tutorial, we used the VM's ID as the executable name to simplify the process. However, AvalancheGo would also accept `timestampvm-rs` or `timestamp-rs` since those are registered aliases in previous step.

Wrapping Up

That's it! That's the entire implementation of a VM which defines a blockchain-based timestamp server written in Rust!

In this tutorial, we learned:

- The `block::ChainVM` trait, which all VMs that define a linear chain must implement
- The `snowman::Block` trait, which all blocks that are part of a linear chain must implement
- The `subnet` mod, which allows blockchains to run in their own processes using the `rpcchainvm`.
- An actual implementation of `block::ChainVM` and `snowman::Block`.

How to Build a Simple VM From Scratch

This is part of a series of tutorials for building a Virtual Machine (VM):

- [Introduction to VMs](#)
- [How to Build a Simple Golang VM](#)
- [How to Build a Complex Golang VM](#)
- [How to Build a Simple Rust VM](#)
- How to Build a Simple VM From Scratch (this article)

Introduction

This is a language-agnostic high-level documentation explaining the basics of how to get started at implementing your own virtual machine from scratch.

Avalanche virtual machines are gRPC servers implementing Avalanche's [Proto interfaces](#). This means that it can be done in [any language that has a gRPC implementation](#).

Minimal Implementation

To get the process started, at the minimum, you will to implement the following interfaces :

- [vm.Runtime](#) (Client)
- [vm.VM](#) (Server)

To build a blockchain taking advantage of AvalancheGo's consensus to build blocks, you will need to implement :

- [AppSender](#) (Client)
- [Messenger](#) (Client)

To have a json-RPC endpoint, `/ext/bc/subnetId/rpc` exposed by AvalancheGo, you will need to implement :

- [Http](#) (Server)

You can and should use a tool like `buf` to generate the (Client/Server) code from the interfaces as stated in the [Avalanche module](#)'s page.

:::note There are *server* and *client* interfaces to implement. AvalancheGo calls the *server* interfaces exposed by your VM and your VM calls the *client* interfaces exposed by AvalancheGo. :::

Starting Process

Your VM is started by AvalancheGo launching your binary. Your binary is started as a sub-process of AvalancheGo. While launching your binary, avalanche's go passes an environment variable `AVALANCHE_VM_RUNTIME_ENGINE_ADDR` containing an url. We must use this url to initialize a `vm.Runtime` client.

Your VM, after having started a grpc server implementing the VM interface must call the [vm.Runtime.InitializeRequest](#) with the following parameters.

- `protocolVersion` : It must match the `supported plugin version` of the [AvalancheGo release](#) you are using. It is always part of the release notes.
- `addr` : It is your grpc server's address. It must be in the following format `host:port` (example `localhost:12345`)

VM Initialization

The service methods are described in the same order as they are called. You will need to implement these methods in your server.

Pre-Initialization Sequence

AvalancheGo starts/stops your process multiple times before launching the real initialization

- [VM.Version](#)
 - Return : your VM's version.
- [VM.CreateStaticHandler](#)
 - Return : an empty array - (Not absolutely required).
- [VM.Shutdown](#)
 - You should gracefully stop your process.
 - Return : Empty

Initialization Sequence

- [VM.CreateStaticHandlers](#)
 - Return an empty array - (Not absolutely required).
- [VM.Initialize](#)
 - Param : an [InitializeRequest](#).
 - You must use this data to initialize your VM.
 - You should add the genesis block to your blockchain and set it as the last accepted block.
 - Return : an [InitializeResponse](#) containing data about the genesis extracted from the `genesis_bytes` that was sent in the request.
- [VM.VerifyHeightIndex](#)
 - Return : a [VerifyHeightIndexResponse](#) with the code `ERROR_UNSPECIFIED` to indicate that no error has occurred.
- [VM.CreateHandlers](#)
 - To serve json-RPC endpoint, `/ext/bc/subnetId/rpc` exposed by AvalancheGo
 - See [json-RPC](#) for more detail
 - Create a [Http](#) server and get its url.
 - Return : a `CreateHandlersResponse` containing a single item with the server's url. (or an empty array if not implementing the json-RPC endpoint)
- [VM.StateSyncEnabled](#)
 - Return : `true` if you want to enable StateSync, `false` otherwise.
- [VM.SetState](#) If you had specified `true` in the `StateSyncEnabled` result
 - Param : a [SetStateRequest](#) with the `StateSyncing` value
 - Set your blockchain's state to `StateSyncing`
 - Return : a [SetStateResponse](#) built from the genesis block.
- [VM.GetOngoingSyncStateSummary](#) If you had specified `true` in the `StateSyncEnabled` result
 - Return : a [GetOngoingSyncStateSummaryResponse](#) built from the genesis block.
- [VM.SetState](#)
 - Param : a [SetStateRequest](#) with the `Bootstrapping` value
 - Set your blockchain's state to `Bootstrapping`
 - Return : a [SetStateResponse](#) built from the genesis block.
- [VM.SetPreference](#)

- Param : `SetPreferenceRequest` containing the preferred block ID
 - Return : Empty
- [VM.SetState](#)
 - Param : a [SetStateRequest](#) with the `NormalOp` value
 - Set your blockchain's state to `NormalOp`
 - Return : a [SetStateResponse](#) built from the genesis block.
- [VM.Connected](#) (for every other node validating this Subnet in the network)
 - Param : a [ConnectedRequest](#) with the NodeID and the version of AvalancheGo.
 - Return : Empty
- [VM.Health](#)
 - Param : Empty
 - Return : a [HealthResponse](#) with an empty `details` property.
- [VM.ParseBlock](#)
 - Param : A byte array containing a Block (the genesis block in this case)
 - Return : a [ParseBlockResponse](#) built from the last accepted block.

At this point, your VM is fully started and initialized.

Building Blocks

Transaction Gossiping Sequence

When your VM receives transactions (for example using the [json-RPC](#) endpoints), it can gossip them to the other nodes by using the [AppSender](#) service.

Supposing we have a 3 nodes network with nodeX, nodeY, nodeZ

NodeX has received a new transaction on its json-RPC endpoint : *on nodeX*

- [AppSender.SendAppGossip \(client\)](#)
 - You must serialize your transaction data into a byte array and call the `SendAppGossip` to propagate the transaction.

AvalancheGo then propagates this to the other nodes.

on nodeY and nodeZ

- [VM.AppGossip](#)
 - Param : A byte array containing your transaction data, and the NodeID of the node which sent the gossip message.
 - You must deserialize the transaction and store it for the next block.
 - Return : Empty

Block Building Sequence

Whenever your VM is ready to build a new block, it will initiate the block building process by using the [Messenger](#) service. Supposing that nodeY wants to build the block. *you probably will implement some kind of background worker checking every second if there are any pending transactions :*

on nodeY

- [client Messenger.Notify](#)
 - You must issue a notify request to AvalancheGo by calling the method with the `MESSAGE_BUILD_BLOCK` value.

on nodeY

- [VM.BuildBlock](#)
 - Param : Empty
 - You must build a block with your pending transactions. Serialize it to a byte array.
 - Store this block in memory as a "pending blocks"
 - Return : a [BuildBlockResponse](#) from the newly built block and its associated data (`id`, `parent_id`, `height`, `timestamp`).
- [VM.BlockVerify](#)
 - Param : The byte array containing the block data
 - Return : the block's timestamp
- [VM.SetPreference](#)
 - Param : The block's ID
 - You must mark this block as the next preferred block.
 - Return : Empty

on nodeX and nodeZ

- [VM.ParseBlock](#)
 - Param : A byte array containing the newly built block's data
 - Store this block in memory as a "pending blocks"
 - Return : a [ParseBlockResponse](#) built from the last accepted block.
- [VM.BlockVerify](#)
 - Param : The byte array containing the block data
 - Return : the block's timestamp
- [VM.SetPreference](#)
 - Param : The block's ID
 - You must mark this block as the next preferred block.
 - Return : Empty

on all nodes

- [VM.BlockAccept](#)
 - Param : The block's ID
 - You must accept this block as your last final block.
 - Return : Empty

Managing Conflicts

Conflicts happen when two or more nodes propose the next block at the same time. AvalancheGo takes care of this and decides which block should be considered final, and which blocks should be rejected using Snowman consensus. On the VM side, all there is to do is implement the `VM.BlockAccept` and `VM.BlockReject` methods.

`nodeX proposes block 0x123... , nodeY proposes block 0x321... and nodeZ proposes block 0x456`

There are three conflicting blocks (different hashes), and if we look at our VM's log files, we can see that AvalancheGo uses Snowman to decide which block must be accepted.

```
...
... snowman/voter.go:58 filtering poll results ...
... snowman/voter.go:65 finishing poll ...
... snowman/voter.go:87 Snowman engine can't quiesce
...
... snowman/voter.go:58 filtering poll results ...
... snowman/voter.go:65 finishing poll ...
... snowman/topological.go:600 accepting block
...
```

Supposing that AvalancheGo accepts block `0x123...`. The following RPC methods are called on all nodes :

- [VM.BlockAccept](#)
 - Param : The block's ID (`0x123...`)
 - You must accept this block as your last final block.
 - Return : Empty
- [VM.BlockReject](#)
 - Param : The block's ID (`0x321...`)
 - You must mark this block as rejected.
 - Return : Empty
- [VM.BlockReject](#)
 - Param : The block's ID (`0x456...`)
 - You must mark this block as rejected.
 - Return : Empty

json-RPC

To enable your json-RPC endpoint, you must implement the `HandleSimple` method of the [Http](#) interface.

- [Http.HandleSimple](#)
 - Param : a [HandleSimpleHTTPRequest](#) containing the original request's method, url, headers, and body.
 - Analyze, deserialize and handle the request
for example, if the request represents a transaction, we must deserialize it, check the signature, store it and gossip it to the other nodes using the messenger client
 - Return the [HandleSimpleHTTPResponse](#) response that will be sent back to the original sender.

This server is registered with AvalancheGo during the [Initialization process](#) when the `VM.CreateHandlers` method is called. You must simply respond with the server's url in the `CreateHandlersResponse` result.

How to Build a Complex Golang VM

This is part of a series of tutorials for building a Virtual Machine (VM):

- [Introduction to Virtual Machines](#)
- [How to Build a Simple Golang VM](#)
- How to Build a Complex Golang VM (this article)
- [How to Build a Simple Rust VM](#)

Introduction

In this tutorial, we'll walk through how to build a virtual machine by referencing [the BlobVM](#). The BlobVM is a virtual machine that can be used to implement a decentralized key-value store.

A blob (shorthand for "binary large object") is an arbitrary piece of data. BlobVM stores a key-value pair by breaking it apart into multiple chunks stored with their hashes as their keys in the blockchain. A root key-value pair has references to these chunks for lookups. By default, the maximum chunk size is set to 200 KiB.

Prerequisites

Make sure you have followed the previous tutorials in this series:

- [Introduction to Virtual Machines](#)
- [How to Build a Simple VM](#)

Components

A VM defines how a blockchain should be built. A block is populated with a set of transactions which mutate the state of the blockchain when executed. When a block with a set of transactions is applied to a given state, a state transition occurs by executing all of the transactions in the block in-order and applying it to the previous block of the blockchain. By executing a series of blocks chronologically, anyone can verify and reconstruct the state of the blockchain at an arbitrary point in time.

The BlobVM repository has a few components to handle the lifecycle of tasks from a transaction being issued to a block being accepted across the network:

- **Transaction** - A state transition
- **Mempool** - Stores pending transactions that haven't been finalized yet
- **Network** - Propagates transactions from the mempool other nodes in the network
- **Block** - Defines the block format, how to verify it, and how it should be accepted or rejected across the network
- **Block Builder** - Builds blocks by including transactions from the mempool
- **Virtual Machine** - Application-level logic. Implements the VM interface needed to interact with Avalanche consensus and defines the blueprint for the blockchain.
- **Service** - Exposes APIs so users can interact with the VM
- **Factory** - Used to initialize the VM

Lifecycle of a Transaction

A VM will often times expose a set of APIs so users can interact with the it. In the blockchain, blocks can contain a set of transactions which mutate the blockchain's state. Let's dive into the lifecycle of a transaction from its issuance to its finalization on the blockchain.

- A user makes an API request to `service.IssueRawTx` to issue their transaction
 - This API will deserialize the user's transaction and forward it to the VM
- The transaction is submitted to the VM
 - The transaction is added to the VM's mempool
- The VM asynchronously periodically gossips new transactions in its mempool to other nodes in the network so they can learn about them
- The VM sends the Avalanche consensus engine a message to indicate that it has transactions in the mempool that are ready to be built into a block
- The VM proposes the block with to consensus
- Consensus verifies that the block is valid and well-formed
- Consensus gets the network to vote on whether the block should be accepted or rejected
 - If a block is rejected, its transactions are reclaimed by the mempool so they can be included in a future block
 - If a block is accepted, it's finalized by writing it to the blockchain

Coding the Virtual Machine

We'll dive into a few of the packages that are in the The BlobVM repository to learn more about how they work:

- `vm`
 - `block_builder.go`
 - `chain_vm.go`
 - `network.go`
 - `service.go`
 - `vm.go`
- `chain`
 - `unsigned_tx.go`
 - `base_tx.go`
 - `transfer_tx.go`
 - `set_tx.go`
 - `tx.go`
 - `block.go`
 - `mempool.go`
 - `storage.go`
 - `builder.go`
- `mempool`
 - `mempool.go`

Transactions

The state the blockchain can only be mutated by getting the network to accept a signed transaction. A signed transaction contains the transaction to be executed alongside the signature of the issuer. The signature is required to cryptographically verify the sender's identity. A VM can define an arbitrary amount of unique transactions types to support different operations on the blockchain. The BlobVM implements two different transactions types:

- [TransferTx](#) - Transfers coins between accounts.
- [SetTx](#) - Stores a key-value pair on the blockchain.

UnsignedTransaction

All transactions in the BlobVM implement the common [UnsignedTransaction](#) interface, which exposes shared functionality for all transaction types.

```
type UnsignedTransaction interface {
    Copy() UnsignedTransaction
    GetBlockID() ids.ID
    GetMagic() uint64
    GetPrice() uint64
    SetBlockID(ids.ID)
    SetMagic(uint64)
    SetPrice(uint64)
    FeeUnits(*Genesis) uint64 // number of units to mine tx
    LoadUnits(*Genesis) uint64 // units that should impact fee rate

    ExecuteBase(*Genesis) error
    Execute(*TransactionContext) error
    TypedData() *tdata.TypedData
    Activity() *Activity
}
```

BaseTx

Common functionality and metadata for transaction types are implemented by [BaseTx](#).

- [SetBlockID](#) sets the transaction's block ID.
- [GetBlockID](#) returns the transaction's block ID.
- [SetMagic](#) sets the magic number. The magic number is used to differentiate chains to prevent replay attacks
- [GetMagic](#) returns the magic number. Magic number is defined in genesis.
- [SetPrice](#) sets the price per fee unit for this transaction.
- [GetPrice](#) returns the price for this transaction.
- [FeeUnits](#) returns the fee units this transaction will consume.
- [LoadUnits](#) identical to [FeeUnits](#)
- [ExecuteBase](#) executes common validation checks across different transaction types. This validates the transaction contains a valid block ID, magic number, and gas price as defined by genesis.

TransferTx

[TransferTx](#) supports the transfer of tokens from one account to another.

```
type TransferTx struct {
    *BaseTx `serialize:"true" json:"baseTx"`

    // To is the recipient of the [Units].
    To common.Address `serialize:"true" json:"to"`

    // Units are transferred to [To].
    Units uint64 `serialize:"true" json:"units"`
}
```

[TransferTx](#) embeds [BaseTx](#) to avoid re-implementing common operations with other transactions, but implements its own [Execute](#) to support token transfers.

This performs a few checks to ensure that the transfer is valid before transferring the tokens between the two accounts.

```
func (t *TransferTx) Execute(c *TransactionContext) error {
    // Must transfer to someone
    if bytes.Equal(t.To[:], zeroAddress[:]) {
        return ErrNonActionable
    }

    // This prevents someone from transferring to themselves.
    if bytes.Equal(t.To[:], c.Sender[:]) {
        return ErrNonActionable
    }
    if t.Units == 0 {
        return ErrNonActionable
    }
    if _, err := ModifyBalance(c.Database, c.Sender, false, t.Units); err != nil {
        return err
    }
    if _, err := ModifyBalance(c.Database, t.To, true, t.Units); err != nil {
        return err
    }
    return nil
}
```

SetTx

`SetTx` is used to assign a value to a key.

```
type SetTx struct {
    *BaseTx `serialize:"true" json:"baseTx"`

    Value []byte `serialize:"true" json:"value"`
}
```

`SetTx` implements its own `FeeUnits` method to compensate the network according to the size of the blob being stored.

```
func (s *SetTx) FeeUnits(g *Genesis) uint64 {
    // We don't subtract by 1 here because we want to charge extra for any
    // value-based interaction (even if it is small or a delete).
    return s.BaseTx.FeeUnits(g) + valueUnits(g, uint64(len(s.Value)))
}
```

`SetTx`'s `Execute` method performs a few safety checks to validate that the blob meets the size constraints enforced by genesis and doesn't overwrite an existing key before writing it to the blockchain.

```
func (s *SetTx) Execute(t *TransactionContext) error {
    g := t.Genesis
    switch {
        case len(s.Value) == 0:
            return ErrValueEmpty
        case uint64(len(s.Value)) > g.MaxValueSize:
            return ErrValueTooBig
    }

    k := ValueHash(s.Value)

    // Do not allow duplicate value setting
    _, exists, err := GetValueMeta(t.Database, k)
    if err != nil {
        return err
    }
    if exists {
        return ErrKeyExists
    }

    return PutKey(t.Database, k, &ValueMeta{
        Size:     uint64(len(s.Value)),
        TxID:    t.TxID,
        Created: t.BlockTime,
    })
}
```

Signed Transaction

The unsigned transactions mentioned previously can't be issued to the network without first being signed. BlobVM implements signed transactions by embedding the unsigned transaction alongside its signature in [Transaction](#). In BlobVM, a signature is defined as the [ECDSA signature](#) of the issuer's private key of the [KECCAK256](#) hash of the unsigned transaction's data ([digest hash](#)).

```
type Transaction struct {
    UnsignedTransaction `serialize:"true" json:"unsignedTransaction"`
    Signature          []byte `serialize:"true" json:"signature"`

    digestHash []byte
    bytes      []byte
    id         ids.ID
    size       uint64
    sender     common.Address
}
```

The `Transaction` type wraps any unsigned transaction. When a `Transaction` is executed, it calls the `Execute` method of the underlying embedded `UnsignedTx` and performs the following sanity checks:

1. The underlying `UnsignedTx` must meet the requirements set by genesis
 1. This includes checks to make sure that the transaction contains the correct magic number and meets the minimum gas price as defined by genesis
2. The transaction's block ID must be a recently accepted block
3. The transaction must not be a recently issued transaction
4. The issuer of the transaction must have enough gas
5. The transaction's gas price must be meet the next expected block's minimum gas price
6. The transaction must execute without error

If the transaction is successfully verified, it's submitted as a pending write to the blockchain.

```

func (t *Transaction) Execute(g *Genesis, db database.Database, blk *StatelessBlock, context *Context) error {
    if err := t.UnsignedTransaction.ExecuteBase(g); err != nil {
        return err
    }
    if !context.RecentBlockIDs.Contains(t.GetBlockID()) {
        // Hash must be recent to be any good
        // Should not happen because of mempool cleanup
        return ErrInvalidBlockID
    }
    if context.RecentTxIDs.Contains(t.ID()) {
        // Tx hash must not be recently executed (otherwise could be replayed)
        //
        // NOTE: We only need to keep cached tx hashes around as long as the
        // block hash referenced in the tx is valid
        return ErrDuplicateTx
    }

    // Ensure sender has balance
    if _, err := ModifyBalance(db, t.sender, false, t.FeeUnits(g)*t.GetPrice()); err != nil {
        return err
    }
    if t.GetPrice() < context.NextPrice {
        return ErrInsufficientPrice
    }
    if err := t.UnsignedTransaction.Execute(&TransactionContext{
        Genesis: g,
        Database: db,
        BlockTime: uint64(blk.Tmstmp),
        TxID: t.id,
        Sender: t.sender,
    }); err != nil {
        return err
    }
    if err := SetTransaction(db, t); err != nil {
        return err
    }
    return nil
}

```

Example

Let's walk through an example on how to issue a `SetTx` transaction to the BlobVM to write a key-value pair.

- Create the unsigned transaction for `SetTx`

```

utx := &chain.SetTx{
    BaseTx: &chain.BaseTx{},
    Value: []byte("data"),
}

utx.SetBlockID(lastAcceptedID)
utx.SetMagic(genesis.Magic)
utx.SetPrice(price + blockCost/utx.FeeUnits(genesis))

```

- Calculate the [digest hash](#) for the transaction.

```
digest, err := chain.DigestHash(utx)
```

- [Sign](#) the digest hash with the issuer's private key.

```
signature, err := chain.Sign(digest, privateKey)
```

- Create and initialize the new signed transaction.

```

tx := chain.NewTx(utx, sig)
if err := tx.Init(g); err != nil {
    return ids.Empty, 0, err
}

```

- Issue the request with the client

```
txID, err = cli.IssueRawTx(ctx, tx.Bytes())
```

Mempool

Mempool Overview

The [mempool](#) is a buffer of volatile memory that stores pending transactions. Transactions are stored in the mempool whenever a node learns about a new transaction either through gossip with other nodes or through an API call issued by a user.

The mempool is implemented as a min-max [heap](#) ordered by each transaction's gas price. The mempool is created during the [initialization](#) of VM.

```
vm.mempool = mempool.New(vm.genesis, vm.config.MempoolSize)
```

Whenever a transaction is submitted to VM, it first gets initialized, verified, and executed locally. If the transaction looks valid, then it's added to the [mempool](#).

```
vm.mempool.Add(tx)
```

Mempool Methods

Add

When a transaction is added to the mempool, [Add](#) is called. This performs the following:

- Checks if the transaction being added already exists in the mempool or not
- The transaction is added to the min-max heap
- If the mempool's heap size is larger than the maximum configured value, then the lowest paying transaction is evicted
- The transaction is added to the list of transactions that are able to be gossiped to other peers
- A notification is sent through the in the `mempool.Pending` channel to indicate that the consensus engine should build a new block

Block Builder

Block Builder Overview

The [TimeBuilder](#) implementation for `BlockBuilder` acts as an intermediary notification service between the mempool and the consensus engine. It serves the following functions:

- Periodically gossips new transactions to other nodes in the network
- Periodically notifies the consensus engine that new transactions from the mempool are ready to be built into blocks

`TimeBuilder` and can exist in 3 states:

- `dontBuild` - There are no transactions in the mempool that are ready to be included in a block
- `building` - The consensus engine has been notified that it should build a block and there are currently transactions in the mempool that are eligible to be included into a block
- `mayBuild` - There are transactions in the mempool that are eligible to be included into a block, but the consensus engine has not been notified yet

Block Builder Methods

Gossip

The [Gossip](#) method initiates the gossip of new transactions from the mempool at periodically as defined by `vm.config.GossipInterval`.

Build

The [Build](#) method consumes transactions from the mempool and signals the consensus engine when it's ready to build a block.

If the mempool signals the `TimeBuilder` that it has available transactions, `TimeBuilder` will signal consensus that the VM is ready to build a block by sending the consensus engine a `common.PendingTxs` message.

When the consensus engine receives the `common.PendingTxs` message it calls the VM's `BuildBlock` method. The VM will then build a block from eligible transactions in the mempool.

- If there are still remaining transactions in the mempool after a block is built, then the `TimeBuilder` is put into the `mayBuild` state to indicate that there are still transactions that are eligible to be built into block, but the consensus engine isn't aware of it yet.

Network

[Network](#) handles the workflow of gossiping transactions from a node's mempool to other nodes in the network.

Network Methods

GossipNewTxs

`GossipNewTxs` sends a list of transactions to other nodes in the network. `TimeBuilder` calls the network's `GossipNewTxs` function to gossip new transactions in the mempool.

```
func (n *PushNetwork) GossipNewTxs(newTxs []*chain.Transaction) error {
    txs := []*chain.Transaction{}

    // Gossip at most the target units of a block at once
    for _, tx := range newTxs {
        if _, exists := n.gossipedTxs.Get(tx.ID()); exists {
            log.Debug("already gossiped, skipping", "txId", tx.ID())
            continue
        }
        n.gossipedTxs.Put(tx.ID(), nil)
        txs = append(txs, tx)
    }
}
```

```

    return n.sendTxs(txs)
}

```

Recently gossiped transactions are maintained in a cache to avoid DDoSing a node from repeated gossip failures.

Other nodes in the network will receive the gossiped transactions through their `AppGossip` handler. Once a gossip message is received, it's deserialized and the new transactions are submitted to the VM.

```

func (vm *VM) AppGossip(nodeID ids.NodeID, msg []byte) error {
    txs := make([]*chain.Transaction, 0)
    if _, err := chain.Unmarshal(msg, &txs); err != nil {
        return nil
    }

    // submit incoming gossip
    log.Debug("AppGossip transactions are being submitted", "txs", len(txs))
    if errs := vm.Submit(txs...); len(errs) > 0 {
        for _, err := range errs {
            log.Error("Error submitting gossip transaction", "err", err)
        }
    }

    return nil
}

```

Block

Blocks go through a lifecycle of being proposed by a validator, verified, and decided by consensus. Upon acceptance, a block will be committed and will be finalized on the blockchain.

BlobVM implements two types of blocks, `StatefulBlock` and `StatelessBlock`.

StatefulBlock

A `StatefulBlock` contains strictly the metadata about the block that needs to be written to the database.

```

type StatefulBlock struct {
    Prnt      ids.ID      `serialize:"true" json:"parent"`
    Tmstmp   int64       `serialize:"true" json:"timestamp"`
    Hght     uint64      `serialize:"true" json:"height"`
    Price    uint64      `serialize:"true" json:"price"`
    Cost     uint64      `serialize:"true" json:"cost"`
    AccessProof common.Hash `serialize:"true" json:"accessProof"`
    Txns     []*Transaction `serialize:"true" json:"txns"`
}

```

StatelessBlock

`StatelessBlock` is a superset of `StatefulBlock` and additionally contains fields that are needed to support block-level operations like verification and acceptance throughout its lifecycle in the VM.

```

type StatelessBlock struct {
    *StatefulBlock `serialize:"true" json:"block"`
    id           ids.ID
    st           choices.Status
    t             time.Time
    bytes        []byte
    vm           VM
    children     []*StatelessBlock
    onAcceptDB   *versiondb.Database
}

```

Let's have a look at the fields of `StatelessBlock`:

- `StatefulBlock` - The metadata about the block that will be written to the database upon acceptance
- `bytes` - The serialized form of the `StatefulBlock`.
- `id` - The Keccak256 hash of `bytes`.
- `st` - The status of the block in consensus (i.e `Processing`, `Accepted`, or `Rejected`)
- `children` - The children of this block
- `onAcceptDB` - The database this block should be written to upon acceptance.

When the consensus engine tries to build a block by calling the VM's `BuildBlock`, the VM calls the `block.NewBlock` function to get a new block that is a child of the currently preferred block.

```

func NewBlock(vm VM, parent snowman.Block, tmstp int64, context *Context) *StatelessBlock {
    return &StatelessBlock{
        StatefulBlock: StatefulBlock{
            Prnt: parent.ID,
            Tmstmp: tmstp,
        },
        id:           id,
        st:           st,
        t:            time.Now(),
        bytes:        bytes,
        onAcceptDB:   db,
    }
}

```

```

        StatefulBlock: &StatefulBlock{
            Tmstmp: tmstp,
            Prnt: parent.ID(),
            Hght: parent.Height() + 1,
            Price: context.NextPrice,
            Cost: context.NextCost,
        },
        vm: vm,
        st: choices.Processing,
    }
}

```

Some `StatelessBlock` fields like the block ID, byte representation, and timestamp aren't populated immediately. These are set during the `StatelessBlock`'s [init](#) method, which initializes these fields once the block has been populated with transactions.

```

func (b *StatelessBlock) init() error {
    bytes, err := Marshal(b.StatefulBlock)
    if err != nil {
        return err
    }
    b.bytes = bytes

    id, err := ids.ToID(crypto.Keccak256(b.bytes))
    if err != nil {
        return err
    }
    b.id = id
    b.t = time.Unix(b.StatefulBlock.Tmstmp, 0)
    g := b.vm.Genesis()
    for _, tx := range b.StatefulBlock.Txs {
        if err := tx.Init(g); err != nil {
            return err
        }
    }
    return nil
}

```

To build the block, the VM will try to remove as many of the highest-paying transactions from the mempool to include them in the new block until the maximum block fee set by genesis is reached.

A block once built, can exist in two states:

- Rejected - The block was not accepted by consensus
 - In this case, the mempool will reclaim the rejected block's transactions so they can be included in a future block.
- Accepted - The block was accepted by consensus
 - In this case, we write the block to the blockchain by committing it to the database.

When the consensus engine receives the built block, it calls the block's [Verify](#) method to validate that the block is well-formed. In BlobVM, the following constraints are placed on valid blocks:

- A block must contain at least one transaction and the block's timestamp must be within 10s into the future.

```

if len(b.Txs) == 0 {
    return nil, nil, ErrNoTxs
}
if b.Timestamp().Unix() >= time.Now().Add(futureBound).Unix() {
    return nil, nil, ErrTimestampTooLate
}

```

- The sum of the gas units consumed by the transactions in the block must not exceed the gas limit defined by genesis.

```

blockSize := uint64(0)
for _, tx := range b.Txs {
    blockSize += tx.LoadUnits(g)
    if blockSize > g.MaxBlockSize {
        return nil, nil, ErrBlockTooBig
    }
}

```

- The parent block of the proposed block must exist and have an earlier timestamp.

```

parent, err := b.vm.GetStatelessBlock(b.Prnt)
if err != nil {
    log.Debug("could not get parent", "id", b.Prnt)
    return nil, nil, err
}

```

```

    }
    if b.Timestamp().Unix() < parent.Timestamp().Unix() {
        return nil, nil, ErrTimestampTooEarly
    }
}

```

- The target block price and minimum gas price must meet the minimum enforced by the VM.

```

context, err := b.vm.ExecutionContext(b.Tmstmp, parent)
if err != nil {
    return nil, nil, err
}
if b.Cost != context.NextCost {
    return nil, nil, ErrInvalidCost
}
if b.Price != context.NextPrice {
    return nil, nil, ErrInvalidPrice
}

```

After the results of consensus are complete, the block is either accepted by committing the block to the database or rejected by returning the block's transactions into the mempool.

```

// implements "snowman.Block.choices.Decidable"
func (b *StatelessBlock) Accept() error {
    if err := b.onAcceptDB.Commit(); err != nil {
        return err
    }
    for _, child := range b.children {
        if err := child.onAcceptDB.SetDatabase(b.vm.State()); err != nil {
            return err
        }
    }
    b.st = choices.Accepted
    b.vm.Accepted(b)
    return nil
}

// implements "snowman.Block.choices.Decidable"
func (b *StatelessBlock) Reject() error {
    b.st = choices.Rejected
    b.vm.Rejected(b)
    return nil
}

```

API

[Service](#) implements an API server so users can interact with the VM. The VM implements the interface method [CreateHandlers](#) that exposes the VM's RPC API.

```

func (vm *VM) CreateHandlers() (map[string]*common.HTTPHandler, error) {
    apis := map[string]*common.HTTPHandler{}
    public, err := newHandler(Name, &PublicService{vm: vm})
    if err != nil {
        return nil, err
    }
    apis[PublicEndpoint] = public
    return apis, nil
}

```

One API that's exposed is [IssueRawTx](#) to allow users to issue transactions to the BlobVM. It accepts an `IssueRawTxArgs` that contains the transaction the user wants to issue and forwards it to the VM.

```

func (svc *PublicService) IssueRawTx(_ *http.Request, args *IssueRawTxArgs, reply *IssueRawTxReply) error {
    tx := new(chain.Transaction)
    if _, err := chain.Unmarshal(args.Tx, tx); err != nil {
        return err
    }

    // otherwise, unexported tx.id field is empty
    if err := tx.Init(svc.vm.genesis); err != nil {
        return err
    }
    reply.TxID = tx.ID()

    errs := svc.vm.Submit(tx)
    if len(errs) == 0 {
        return nil
    }
}

```

```

        }
        if len(errs) == 1 {
            return errs[0]
        }
        return fmt.Errorf("%v", errs)
    }
}

```

Virtual Machine

We have learned about all the components used in the BlobVM. Most of these components are referenced in the `vm.go` file, which acts as the entry point for the consensus engine as well as users interacting with the blockchain. For example, the engine calls `vm.BuildBlock()`, that in turn calls `chain.BuildBlock()`. Another example is when a user issues a raw transaction through service APIs, the `vm.Submit()` method is called.

Let's look at some of the important methods of `vm.go` that must be implemented:

Virtual Machine Methods

Initialize

`Initialize` is invoked by `avalanchego` when creating the blockchain. `avalanchego` passes some parameters to the implementing VM.

- `ctx` - Metadata about the VM's execution
- `dbManager` - The database that the VM can write to
- `genesisBytes` - The serialized representation of the genesis state of this VM
- `upgradeBytes` - The serialized representation of network upgrades
- `configBytes` - The serialized VM-specific [configuration](#)
- `toEngine` - The channel used to send messages to the consensus engine
- `fxs` - Feature extensions that attach to this VM
- `appSender` - Used to send messages to other nodes in the network

BlobVM upon initialization persists these fields in its own state to use them throughout the lifetime of its execution.

```

// implements "snowmanblock.ChainVM.common.VM"
func (vm *VM) Initialize(
    ctx *snow.Context,
    dbManager manager.Manager,
    genesisBytes []byte,
    upgradeBytes []byte,
    configBytes []byte,
    toEngine chan<- common.Message,
    _ []*common.Fx,
    appSender common.AppSender,
) error {
    log.Info("initializing blobvm", "version", version.Version)

    // Load config
    vm.config.SetDefaults()
    if len(configBytes) > 0 {
        if err := json.Unmarshal(configBytes, &vm.config); err != nil {
            return fmt.Errorf("failed to unmarshal config %s: %w", string(configBytes), err)
        }
    }

    vm.ctx = ctx
    vm.db = dbManager.Current().Database
    vm.activityCache = make([]*chain.Activity, vm.config.ActivityCacheSize)

    // Init channels before initializing other structs
    vm.stop = make(chan struct{})
    vm.builderStop = make(chan struct{})
    vm.doneBuild = make(chan struct{})
    vm.doneGossip = make(chan struct{})
    vm.appSender = appSender
    vm.network = vm.NewPushNetwork()

    vm.blocks = &cache.LRU{Size: blocksLRUSize}
    vm.verifiedBlocks = make(map[ids.ID]*chain.StatelessBlock)

    vm.toEngine = toEngine
    vm.builder = vm.NewTimeBuilder()

    // Try to load last accepted
    has, err := chain.HasLastAccepted(vm.db)
    if err != nil {
        log.Error("could not determine if have last accepted")
        return err
    }

    // Parse genesis data
}

```

```

vm.genesis = new(chain.Genesis)
if err := ejson.Unmarshal(genesisBytes, vm.genesis); err != nil {
    log.Error("could not unmarshal genesis bytes")
    return err
}
if err := vm.genesis.Verify(); err != nil {
    log.Error("genesis is invalid")
    return err
}
targetUnitsPerSecond := vm.genesis.TargetBlockSize / uint64(vm.genesis.TargetBlockRate)
vm.targetRangeUnits = targetUnitsPerSecond * uint64(vm.genesis.LookbackWindow)
log.Debug("loaded genesis", "genesis", string(genesisBytes), "target range units", vm.targetRangeUnits)

vm.mempool = mempool.New(vm.genesis, vm.config.MempoolSize)

if has { //nolint:nestif
    blkID, err := chain.GetLastAccepted(vm.db)
    if err != nil {
        log.Error("could not get last accepted", "err", err)
        return err
    }

    blk, err := vm.GetStatelessBlock(blkID)
    if err != nil {
        log.Error("could not load last accepted", "err", err)
        return err
    }

    vm.preferred, vm.lastAccepted = blkID, blk
    log.Info("initialized blobvm from last accepted", "block", blkID)
} else {
    genesisBlk, err := chain.ParseStatefulBlock(
        vm.genesis.StatefulBlock(),
        nil,
        choices.Accepted,
        vm,
    )
    if err != nil {
        log.Error("unable to init genesis block", "err", err)
        return err
    }

    // Set Balances
    if err := vm.genesis.Load(vm.db, vm.AirdropData); err != nil {
        log.Error("could not set genesis allocation", "err", err)
        return err
    }

    if err := chain.SetLastAccepted(vm.db, genesisBlk); err != nil {
        log.Error("could not set genesis as last accepted", "err", err)
        return err
    }
    gBlkID := genesisBlk.ID()
    vm.preferred, vm.lastAccepted = gBlkID, genesisBlk
    log.Info("initialized blobvm from genesis", "block", gBlkID)
}
vm.AirdropData = nil

```

After initializing its own state, BlobVM also starts asynchronous workers to build blocks and gossip transactions to the rest of the network.

```

go vm.builder.Build()
go vm.builder.Gossip()
return nil
}

```

GetBlock

[GetBlock](#) returns the block with the provided ID.

GetBlock will attempt to fetch the given block from the database, and return an non-nil error if it wasn't able to get it.

```

func (vm *VM) GetBlock(id ids.ID) (snowman.Block, error) {
    b, err := vm.GetStatelessBlock(id)
    if err != nil {
        log.Warn("failed to get block", "err", err)
    }
}

```

```
    return b, err
}
```

ParseBlock

[ParseBlock](#) deserializes a block.

```
func (vm *VM) ParseBlock(source []byte) (snowman.Block, error) {
    newBlk, err := chain.ParseBlock(
        source,
        choices.Processing,
        vm,
    )
    if err != nil {
        log.Error("could not parse block", "err", err)
        return nil, err
    }
    log.Debug("parsed block", "id", newBlk.ID())

    // If we have seen this block before, return it with the most
    // up-to-date info
    if oldBlk, err := vm.GetBlock(newBlk.ID()); err == nil {
        log.Debug("returning previously parsed block", "id", oldBlk.ID())
        return oldBlk, nil
    }

    return newBlk, nil
}
```

BuildBlock

Avalanche consensus calls [BuildBlock](#) when it receives a notification from the VM that it has pending transactions that are ready to be issued into a block.

```
func (vm *VM) BuildBlock() (snowman.Block, error) {
    log.Debug("BuildBlock triggered")
    blk, err := chain.BuildBlock(vm, vm.preferred)
    vm.builder.HandleGenerateBlock()
    if err != nil {
        log.Debug("BuildBlock failed", "error", err)
        return nil, err
    }
    sblk, ok := blk.(*chain.StatelessBlock)
    if !ok {
        return nil, fmt.Errorf("unexpected snowman.Block %T, expected *StatelessBlock", blk)
    }

    log.Debug("BuildBlock success", "blkID", blk.ID(), "txs", len(sblk.Txs))
    return blk, nil
}
```

SetPreference

[SetPreference](#) sets the block ID preferred by this node. A node votes to accept or reject a block based on its current preference in consensus.

```
func (vm *VM) SetPreference(id ids.ID) error {
    log.Debug("set preference", "id", id)
    vm.preferred = id
    return nil
}
```

LastAccepted

[LastAccepted](#) returns the block ID of the block that was most recently accepted by this node.

```
func (vm *VM) LastAccepted() (ids.ID, error) {
    return vm.lastAccepted.ID(), nil
}
```

CLI

BlobVM implements a generic key-value store, but support to read and write arbitrary files into the BlobVM blockchain is implemented in the [blob-cli](#)

To write a file, BlobVM breaks apart an arbitrarily large file into many small chunks. Each chunk is submitted to the VM in a `SetTx`. A root key is generated which contains all of the hashes of the chunks.

```
func Upload(
    ctx context.Context, cli client.Client, priv *ecdsa.PrivateKey,
    f io.Reader, chunkSize int,
```

```

) (common.Hash, error) {
    hashes := []common.Hash{}
    chunk := make([]byte, chunkSize)
    shouldExit := false
    opts := []client.OpOption{client.WithPollTx()}
    totalCost := uint64(0)
    uploaded := map[common.Hash]struct{}{}
    for !shouldExit {
        read, err := f.Read(chunk)
        if errors.Is(err, io.EOF) || read == 0 {
            break
        }
        if err != nil {
            return common.Hash{}, fmt.Errorf("%w: read error", err)
        }
        if read < chunkSize {
            shouldExit = true
            chunk = chunk[:read]

            // Use small file optimization
            if len(hashes) == 0 {
                break
            }
        }
        k := chain.ValueHash(chunk)
        if _, ok := uploaded[k]; ok {
            color.Yellow("already uploaded k=%s, skipping", k)
        } else if exists, _, err := cli.Resolve(ctx, k); err == nil && exists {
            color.Yellow("already on-chain k=%s, skipping", k)
            uploaded[k] = struct{}{}
        } else {
            tx := &chain.SetTx{
                BaseTx: &chain.BaseTx{},
                Value:  chunk,
            }
            txID, cost, err := client.SignIssueRawTx(ctx, cli, tx, priv, opts...)
            if err != nil {
                return common.Hash{}, err
            }
            totalCost += cost
            color.Yellow("uploaded k=%s txID=%s cost=%d totalCost=%d", k, txID, cost, totalCost)
            uploaded[k] = struct{}{}
        }
        hashes = append(hashes, k)
    }

    r := &Root{}
    if len(hashes) == 0 {
        if len(chunk) == 0 {
            return common.Hash{}, ErrEmpty
        }
        r.Contents = chunk
    } else {
        r.Children = hashes
    }

    rb, err := json.Marshal(r)
    if err != nil {
        return common.Hash{}, err
    }
    rk := chain.ValueHash(rb)
    tx := &chain.SetTx{
        BaseTx: &chain.BaseTx{},
        Value:  rb,
    }
    txID, cost, err := client.SignIssueRawTx(ctx, cli, tx, priv, opts...)
    if err != nil {
        return common.Hash{}, err
    }
    totalCost += cost
    color.Yellow("uploaded root=%v txID=%s cost=%d totalCost=%d", rk, txID, cost, totalCost)
    return rk, nil
}

```

Example 1

```
blob-cli set-file ~/Downloads/computer.gif -> 6fe5a52f52b34fb1e07ba90bad47811c645176d0d49ef0c7a7b4b22013f676c8
```

Given the root hash, a file can be looked up by deserializing all of its children chunk values and reconstructing the original file.

```
// TODO: make multi-threaded
func Download(ctx context.Context, cli client.Client, root common.Hash, f io.Writer) error {
    exists, rb, _, err := cli.Resolve(ctx, root)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("%w:%v", ErrMissing, root)
    }
    var r Root
    if err := json.Unmarshal(rb, &r); err != nil {
        return err
    }

    // Use small file optimization
    if contentLen := len(r.Contents); contentLen > 0 {
        if _, err := f.Write(r.Contents); err != nil {
            return err
        }
        color.Yellow("downloaded root=%v size=%fKB", root, float64(contentLen)/units.KiB)
        return nil
    }

    if len(r.Children) == 0 {
        return ErrEmpty
    }

    amountDownloaded := 0
    for _, h := range r.Children {
        exists, b, _, err := cli.Resolve(ctx, h)
        if err != nil {
            return err
        }
        if !exists {
            return fmt.Errorf("%w:%s", ErrMissing, h)
        }
        if _, err := f.Write(b); err != nil {
            return err
        }
        size := len(b)
        color.Yellow("downloaded chunk=%v size=%fKB", h, float64(size)/units.KiB)
        amountDownloaded += size
    }
    color.Yellow("download complete root=%v size=%fMB", root, float64(amountDownloaded)/units.MiB)
    return nil
}
```

Example 2

```
blob-cli resolve-file 6fe5a52f52b34fb1e07ba90bad47811c645176d0d49ef0c7a7b4b22013f676c8 computer_copy.gif
```

Conclusion

This documentation covers concepts about Virtual Machine by walking through a VM that implements a decentralized key-value store.

You can learn more about the BlobVM by referencing the [README](#) in the GitHub repository.

How to Build a Simple Golang VM

This is part of a series of tutorials for building a Virtual Machine (VM):

- [Introduction to VMs](#)
- How to Build a Simple Golang VM (this article)
- [How to Build a Complex Golang VM](#)
- [How to Build a Simple Rust VM](#)

Introduction

In this tutorial, we'll create a very simple VM called the [TimestampVM](#). Each block in the TimestampVM's blockchain contains a strictly increasing timestamp when the block was created and a 32-byte payload of data.

Such a server is useful because it can be used to prove a piece of data existed at the time the block was created. Suppose you have a book manuscript, and you want to be able to prove in the future that the manuscript exists today. You can add a block to the blockchain where the block's payload is a hash of your manuscript. In the future, you can prove that the manuscript existed today by showing that the block has the hash of your manuscript in its payload (this follows from the fact that finding the pre-image of a hash is impossible).

Prerequisites

Make sure you're familiar with the previous tutorial in this series, which dives into what virtual machines are.

- [Introduction to VMs](#)

TimestampVM Implementation

Now we know the interface our VM must implement and the libraries we can use to build a VM.

Let's write our VM, which implements `block.ChainVM` and whose blocks implement `snowman.Block`. You can also follow the code in the [TimestampVM repository](#).

Codec

`Codec` is required to encode/decode the block into byte representation. TimestampVM uses the default codec and manager.

```
const (
    // CodecVersion is the current default codec version
    CodecVersion = 0
)

// Codecs do serialization and deserialization
var (
    Codec codec.Manager
)

func init() {
    // Create default codec and manager
    c := linearcodec.NewDefault()
    Codec = codec.NewDefaultManager()

    // Register codec to manager with CodecVersion
    if err := Codec.RegisterCodec(CodecVersion, c); err != nil {
        panic(err)
    }
}
```

State

The `State` interface defines the database layer and connections. Each VM should define their own database methods. `State` embeds the `BlockState` which defines block-related state operations.

```
var (
    // These are prefixes for db keys.
    // It's important to set different prefixes for each separate database objects.
    singletonStatePrefix = []byte("singleton")
    blockStatePrefix     = []byte("block")

    _ State = &state{}
)

// State is a wrapper around avax.SingleTonState and BlockState
// State also exposes a few methods needed for managing database commits and close.
type State interface {
    // SingletonState is defined in avalanchego,
    // it is used to understand if db is initialized already.
    avax.SingletonState
    BlockState

    Commit() error
    Close() error
}

type state struct {
    avax.SingletonState
    BlockState

    baseDB *versiondb.Database
}

func NewState(db database.Database, vm *VM) State {
    // create a new baseDB
```

```

baseDB := versiondb.New(db)

// create a prefixed "blockDB" from baseDB
blockDB := prefixdb.New(blockStatePrefix, baseDB)
// create a prefixed "singletonDB" from baseDB
singletonDB := prefixdb.New(singletonStatePrefix, baseDB)

// return state with created sub state components
return &state{
    BlockState:    NewBlockState(blockDB, vm),
    SingletonState: avax.NewSingletonState(singletonDB),
    baseDB:        baseDB,
}
}

// Commit commits pending operations to baseDB
func (s *state) Commit() error {
    return s.baseDB.Commit()
}

// Close closes the underlying base database
func (s *state) Close() error {
    return s.baseDB.Close()
}

```

Block State

This interface and implementation provides storage functions to VM to store and retrieve blocks.

```

const (
    lastAcceptedByte byte = iota
)

const (
    // maximum block capacity of the cache
    blockCacheSize = 8192
)

// persists lastAccepted block IDs with this key
var lastAcceptedKey = []byte{lastAcceptedByte}

var _ BlockState = &blockState{}

// BlockState defines methods to manage state with Blocks and LastAcceptedIDs.
type BlockState interface {
    GetBlock(blkID ids.ID) (*Block, error)
    PutBlock(blk *Block) error

    GetLastAccepted() (ids.ID, error)
    SetLastAccepted(ids.ID) error
}

// blockState implements BlocksState interface with database and cache.
type blockState struct {
    // cache to store blocks
    blkCache cache.Cacher
    // block database
    blockDB   database.Database
    lastAccepted ids.ID

    // vm reference
    vm *VM
}

// blkWrapper wraps the actual blk bytes and status to persist them together
type blkWrapper struct {
    Blk      []byte `serialize:"true"`
    Status  choices.Status `serialize:"true"`
}

// NewBlockState returns BlockState with a new cache and given db
func NewBlockState(db database.Database, vm *VM) BlockState {
    return &blockState{
        blkCache: &cache.LRU{Size: blockCacheSize},
        blockDB:  db,
        vm:       vm,
    }
}

```

```

// GetBlock gets Block from either cache or database
func (s *blockState) GetBlock(blkID ids.ID) (*Block, error) {
    // Check if cache has this blkID
    if blkIntf, cached := s.blkCache.Get(blkID); cached {
        // there is a key but value is nil, so return an error
        if blkIntf == nil {
            return nil, database.ErrNotFound
        }
        // We found it return the block in cache
        return blkIntf.(*Block), nil
    }

    // get block bytes from db with the blkID key
    wrappedBytes, err := s.blockDB.Get(blkID[:])
    if err != nil {
        // we could not find it in the db, let's cache this blkID with nil value
        // so next time we try to fetch the same key we can return error
        // without hitting the database
        if err == database.ErrNotFound {
            s.blkCache.Put(blkID, nil)
        }
        // could not find the block, return error
        return nil, err
    }

    // first decode/unmarshal the block wrapper so we can have status and block bytes
    blkw := blkWrapper{}
    if _, err := Codec.Unmarshal(wrappedBytes, &blkw); err != nil {
        return nil, err
    }

    // now decode/unmarshal the actual block bytes to block
    blk := &Block{}
    if _, err := Codec.Unmarshal(blkw.Blk, blk); err != nil {
        return nil, err
    }

    // initialize block with block bytes, status and vm
    blk.Initialize(blkw.Blk, blkw.Status, s.vm)

    // put block into cache
    s.blkCache.Put(blkID, blk)

    return blk, nil
}

// PutBlock puts block into both database and cache
func (s *blockState) PutBlock(blk *Block) error {
    // create block wrapper with block bytes and status
    blkw := blkWrapper{
        Blk:     blk.Bytes(),
        Status: blk.Status(),
    }

    // encode block wrapper to its byte representation
    wrappedBytes, err := Codec.Marshal(CodecVersion, &blkw)
    if err != nil {
        return err
    }

    blkID := blk.ID()
    // put actual block to cache, so we can directly fetch it from cache
    s.blkCache.Put(blkID, blk)

    // put wrapped block bytes into database
    return s.blockDB.Put(blkID[:], wrappedBytes)
}

// DeleteBlock deletes block from both cache and database
func (s *blockState) DeleteBlock(blkID ids.ID) error {
    s.blkCache.Put(blkID, nil)
    return s.blockDB.Delete(blkID[:])
}

// GetLastAccepted returns last accepted block ID
func (s *blockState) GetLastAccepted() (ids.ID, error) {
    // check if we already have lastAccepted ID in state memory
}

```

```

if s.lastAccepted != ids.Empty {
    return s.lastAccepted, nil
}

// get lastAccepted bytes from database with the fixed lastAcceptedKey
lastAcceptedBytes, err := s.blockDB.Get(lastAcceptedKey)
if err != nil {
    return ids.ID{}, err
}
// parse bytes to ID
lastAccepted, err := ids.ToID(lastAcceptedBytes)
if err != nil {
    return ids.ID{}, err
}
// put lastAccepted ID into memory
s.lastAccepted = lastAccepted
return lastAccepted, nil
}

// SetLastAccepted persists lastAccepted ID into both cache and database
func (s *blockState) SetLastAccepted(lastAccepted ids.ID) error {
    // if the ID in memory and the given memory are same don't do anything
    if s.lastAccepted == lastAccepted {
        return nil
    }
    // put lastAccepted ID to memory
    s.lastAccepted = lastAccepted
    // persist lastAccepted ID to database with fixed lastAcceptedKey
    return s.blockDB.Put(lastAcceptedKey, lastAccepted[:])
}

```

Block

Let's look at our block implementation.

The type declaration is:

```

// Block is a block on the chain.
// Each block contains:
// 1) ParentID
// 2) Height
// 3) Timestamp
// 4) A piece of data (a string)
type Block struct {
    PrntID ids.ID      `serialize:"true" json:"parentID"` // parent's ID
    Hght   uint64       `serialize:"true" json:"height"`   // This block's height. The genesis block is at height 0.
    Tmstmp int64        `serialize:"true" json:"timestamp"` // Time this block was proposed at
    Dt     [dataLen]byte `serialize:"true" json:"data"`      // Arbitrary data

    id     ids.ID      // hold this block's ID
    bytes []byte        // this block's encoded bytes
    status choices.Status // block's status
    vm     *VM          // the underlying VM reference, mostly used for state
}

```

The `serialize:"true"` tag indicates that the field should be included in the byte representation of the block used when persisting the block or sending it to other nodes.

Verify

This method verifies that a block is valid and stores it in the memory. It is important to store the verified block in the memory and return them in the `vm.GetBlock` method.

```

// Verify returns nil iff this block is valid.
// To be valid, it must be that:
// b.parent.Timestamp < b.Timestamp <= [local time] + 1 hour
func (b *Block) Verify() error {
    // Get [b]'s parent
    parentID := b.Parent()
    parent, err := b.vm.getBlock(parentID)
    if err != nil {
        return errDatabaseGet
    }

    // Ensure [b]'s height comes right after its parent's height
    if expectedHeight := parent.Height() + 1; expectedHeight != b.Hght {
        return fmt.Errorf(
            "expected block to have height %d, but found %d",

```

```

        expectedHeight,
        b.Height,
    )
}

// Ensure [b]'s timestamp is after its parent's timestamp.
if b.Timestamp().Unix() < parent.Timestamp().Unix() {
    return errTimestampTooEarly
}

// Ensure [b]'s timestamp is not more than an hour
// ahead of this node's time
if b.Timestamp().Unix() >= time.Now().Add(time.Hour).Unix() {
    return errTimestampTooLate
}

// Put that block to verified blocks in memory
b.vm.verifiedBlocks[b.ID()] = b

return nil
}

```

Accept

`Accept` is called by the consensus to indicate this block is accepted.

```

// Accept sets this block's status to Accepted and sets lastAccepted to this
// block's ID and saves this info to b.vm.DB
func (b *Block) Accept() error {
    b.setStatus(choices.Accepted) // Change state of this block
    blkID := b.ID()

    // Persist data
    if err := b.vm.state.PutBlock(b); err != nil {
        return err
    }

    // Set last accepted ID to this block ID
    if err := b.vm.state.SetLastAccepted(blkID); err != nil {
        return err
    }

    // Delete this block from verified blocks as it's accepted
    delete(b.vm.verifiedBlocks, b.ID())

    // Commit changes to database
    return b.vm.state.Commit()
}

```

Reject

`Reject` is called by the consensus to indicate this block is rejected.

```

// Reject sets this block's status to Rejected and saves the status in state
// Recall that b.vm.DB.Commit() must be called to persist to the DB
func (b *Block) Reject() error {
    b.setStatus(choices.Rejected) // Change state of this block
    if err := b.vm.state.PutBlock(b); err != nil {
        return err
    }

    // Delete this block from verified blocks as it's rejected
    delete(b.vm.verifiedBlocks, b.ID())
    // Commit changes to database
    return b.vm.state.Commit()
}

```

Block Field Methods

These methods are required by the `snowman.Block` interface.

```

// ID returns the ID of this block
func (b *Block) ID() ids.ID { return b.id }

// ParentID returns [b]'s parent's ID
func (b *Block) Parent() ids.ID { return b.PrentID }

// Height returns this block's height. The genesis block has height 0.

```

```

func (b *Block) Height() uint64 { return b.Height }

// Timestamp returns this block's time. The genesis block has time 0.
func (b *Block) Timestamp() time.Time { return time.Unix(b.Timestamp, 0) }

// Status returns the status of this block
func (b *Block) Status() choices.Status { return b.Status }

// Bytes returns the byte repr. of this block
func (b *Block) Bytes() []byte { return b.Bytes }

```

Helper Functions

These methods are convenience methods for blocks, they're not a part of the block interface.

```

// Initialize sets [b.bytes] to [bytes], [b.id] to hash([b.bytes]),
// [b.status] to [status] and [b.vm] to [vm]
func (b *Block) Initialize(bytes []byte, status choices.Status, vm *VM) {
    b.bytes = bytes
    b.id = hashing.ComputeHash256Array(b.bytes)
    b.status = status
    b.vm = vm
}

// SetStatus sets the status of this block
func (b *Block) SetStatus(status choices.Status) { b.status = status }

```

Virtual Machine

Now, let's look at our timestamp VM implementation, which implements the `block.ChainVM` interface.

The declaration is:

```

// This Virtual Machine defines a blockchain that acts as a timestamp server
// Each block contains data (a payload) and the timestamp when it was created

const (
    dataLen = 32
    Name     = "timestampvm"
)

// VM implements the snowman.VM interface
// Each block in this chain contains a Unix timestamp
// and a piece of data (a string)
type VM struct {
    // The context of this vm
    ctx      *snow.Context
    dbManager manager.Manager

    // State of this VM
    state State

    // ID of the preferred block
    preferred ids.ID

    // channel to send messages to the consensus engine
    toEngine chan<- common.Message

    // Proposed pieces of data that haven't been put into a block and proposed yet
    mempool []dataLen]byte

    // Block ID --> Block
    // Each element is a block that passed verification but
    // hasn't yet been accepted/rejected
    verifiedBlocks map[ids.ID]*Block
}

```

Initialize

This method is called when a new instance of VM is initialized. Genesis block is created under this method.

```

// Initialize this vm
// [ctx] is this vm's context
// [dbManager] is the manager of this vm's database
// [toEngine] is used to notify the consensus engine that new blocks are
//   ready to be added to consensus
// The data in the genesis block is [genesisData]

```

```

func (vm *VM) Initialize(
    ctx *snow.Context,
    dbManager manager.Manager,
    genesisData []byte,
    upgradeData []byte,
    configData []byte,
    toEngine chan<- common.Message,
    _ []*common.Fx,
    _ common.AppSender,
) error {
    version, err := vm.Version()
    if err != nil {
        log.Error("error initializing Timestamp VM: %v", err)
        return err
    }
    log.Info("Initializing Timestamp VM", "Version", version)

    vm.dbManager = dbManager
    vm.ctx = ctx
    vm.toEngine = toEngine
    vm.verifiedBlocks = make(map[ids.ID]*Block)

    // Create new state
    vm.state = NewState(vm.dbManager.Current().Database, vm)

    // Initialize genesis
    if err := vm.initGenesis(genesisData); err != nil {
        return err
    }

    // Get last accepted
    lastAccepted, err := vm.state.GetLastAccepted()
    if err != nil {
        return err
    }

    ctx.Log.Info("initializing last accepted block as %s", lastAccepted)

    // Build off the most recently accepted block
    return vm.SetPreference(lastAccepted)
}

```

initGenesis

`initGenesis` is a helper method which initializes the genesis block from given bytes and puts into the state.

```

// Initializes Genesis if required
func (vm *VM) initGenesis(genesisData []byte) error {
    stateInitialized, err := vm.state.IsInitialized()
    if err != nil {
        return err
    }

    // if state is already initialized, skip init genesis.
    if stateInitialized {
        return nil
    }

    if len(genesisData) > dataLen {
        return errBadGenesisBytes
    }

    // genesisData is a byte slice but each block contains an byte array
    // Take the first [dataLen] bytes from genesisData and put them in an array
    var genesisDataArr [dataLen]byte
    copy(genesisDataArr[:], genesisData)

    // Create the genesis block
    // Timestamp of genesis block is 0. It has no parent.
    genesisBlock, err := vm.NewBlock(ids.Empty, 0, genesisDataArr, time.Unix(0, 0))
    if err != nil {
        log.Error("error while creating genesis block: %v", err)
        return err
    }

    // Put genesis block to state
    if err := vm.state.PutBlock(genesisBlock); err != nil {
        log.Error("error while saving genesis block: %v", err)
    }
}

```

```

        return err
    }

    // Accept the genesis block
    // Sets [vm.lastAccepted] and [vm.preferred]
    if err := genesisBlock.Accept(); err != nil {
        return fmt.Errorf("error accepting genesis block: %w", err)
    }

    // Mark this vm's state as initialized, so we can skip initGenesis in further restarts
    if err := vm.state.SetInitialized(); err != nil {
        return fmt.Errorf("error while setting db to initialized: %w", err)
    }

    // Flush VM's database to underlying db
    return vm.state.Commit()
}

```

CreateHandlers

Registered handlers defined in `Service`. See [below](#) for more on APIs.

```

// CreateHandlers returns a map where:
// Keys: The path extension for this blockchain's API (empty in this case)
// Values: The handler for the API
// In this case, our blockchain has only one API, which we name timestamp,
// and it has no path extension, so the API endpoint:
// [Node IP]/ext/bc/[this blockchain's ID]
// See API section in documentation for more information
func (vm *VM) CreateHandlers() (map[string]*common.HTTPHandler, error) {
    server := rpc.NewServer()
    server.RegisterCodec(json.NewCodec(), "application/json")
    server.RegisterCodec(json.NewCodec(), "application/json;charset=UTF-8")
    // Name is "timestamppv"
    if err := server.RegisterService(&Service{vm: vm}, Name); err != nil {
        return nil, err
    }

    return map[string]*common.HTTPHandler{
        "": {
            Handler: server,
        },
    }, nil
}

```

CreateStaticHandlers

Registers static handlers defined in `StaticService`. See [below](#) for more on static APIs.

```

// CreateStaticHandlers returns a map where:
// Keys: The path extension for this VM's static API
// Values: The handler for that static API
func (vm *VM) CreateStaticHandlers() (map[string]*common.HTTPHandler, error) {
    server := rpc.NewServer()
    server.RegisterCodec(json.NewCodec(), "application/json")
    server.RegisterCodec(json.NewCodec(), "application/json;charset=UTF-8")
    if err := server.RegisterService(&StaticService{}, Name); err != nil {
        return nil, err
    }

    return map[string]*common.HTTPHandler{
        "": {
            LockOptions: common.NoLock,
            Handler:     server,
        },
    }, nil
}

```

BuildBlock

`BuildBlock` builds a new block and returns it. This is mainly requested by the consensus engine.

```

// BuildBlock returns a block that this vm wants to add to consensus
func (vm *VM) BuildBlock() (snowman.Block, error) {
    if len(vm.mempool) == 0 { // There is no block to be built
        return nil, errNoPendingBlocks
    }
}

```

```

// Get the value to put in the new block
value := vm.mempool[0]
vm.mempool = vm.mempool[1:]

// Notify consensus engine that there are more pending data for blocks
// (if that is the case) when done building this block
if len(vm.mempool) > 0 {
    defer vm.NotifyBlockReady()
}

// Gets Preferred Block
preferredBlock, err := vm.getBlock(vm.preferred)
if err != nil {
    return nil, fmt.Errorf("couldn't get preferred block: %w", err)
}
preferredHeight := preferredBlock.Height()

// Build the block with preferred height
newBlock, err := vm.NewBlock(vm.preferred, preferredHeight+1, value, time.Now())
if err != nil {
    return nil, fmt.Errorf("couldn't build block: %w", err)
}

// Verifies block
if err := newBlock.Verify(); err != nil {
    return nil, err
}
return newBlock, nil
}

```

NotifyBlockReady

`NotifyBlockReady` is a helper method that can send messages to the consensus engine through `toEngine` channel.

```

// NotifyBlockReady tells the consensus engine that a new block
// is ready to be created
func (vm *VM) NotifyBlockReady() {
    select {
    case vm.toEngine <- common.PendingTxs:
        default:
            vm.ctx.Log.Debug("dropping message to consensus engine")
    }
}

```

GetBlock

`GetBlock` returns the block with the given block ID.

```

// GetBlock implements the snowman.ChainVM interface
func (vm *VM) GetBlock(blkID ids.ID) (snowman.Block, error) { return vm.getBlock(blkID) }

func (vm *VM) getBlock(blkID ids.ID) (*Block, error) {
    // If block is in memory, return it.
    if blk, exists := vm.verifiedBlocks[blkID]; exists {
        return blk, nil
    }

    return vm.state.GetBlock(blkID)
}

```

proposeBlock

This method adds a piece of data to the mempool and notifies the consensus layer of the blockchain that a new block is ready to be built and voted on. This is called by API method `ProposeBlock`, which we'll see later.

```

// proposeBlock appends [data] to [p.mempool].
// Then it notifies the consensus engine
// that new block is ready to be added to consensus
// (namely, a block with data [data])
func (vm *VM) proposeBlock(data [dataLen]byte) {
    vm.mempool = append(vm.mempool, data)
    vm.NotifyBlockReady()
}

```

ParseBlock

Parse a block from its byte representation.

```
// ParseBlock parses [bytes] to a snowman.Block
// This function is used by the vm's state to unmarshal blocks saved in state
// and by the consensus layer when it receives the byte representation of a block
// from another node
func (vm *VM) ParseBlock(bytes []byte) (snowman.Block, error) {
    // A new empty block
    block := &Block{}

    // Unmarshal the byte repr. of the block into our empty block
    _, err := Codec.Unmarshal(bytes, block)
    if err != nil {
        return nil, err
    }

    // Initialize the block
    block.Initialize(bytes, choices.Processing, vm)

    if blk, err := vm.getBlock(block.ID()); err == nil {
        // If we have seen this block before, return it with the most up-to-date
        // info
        return blk, nil
    }

    // Return the block
    return block, nil
}
```

NewBlock

`NewBlock` creates a new block with given block parameters.

```
// NewBlock returns a new Block where:
// - the block's parent is [parentID]
// - the block's data is [data]
// - the block's timestamp is [timestamp]
func (vm *VM) NewBlock(parentID ids.ID, height uint64, data [dataLen]byte, timestamp time.Time) (*Block, error) {
    block := &Block{
        PrntID: parentID,
        Hght:   height,
        Tmstmp: timestamp.Unix(),
        Dt:      data,
    }

    // Get the byte representation of the block
    blockBytes, err := Codec.Marshal(CodecVersion, block)
    if err != nil {
        return nil, err
    }

    // Initialize the block by providing it with its byte representation
    // and a reference to this VM
    block.Initialize(blockBytes, choices.Processing, vm)
    return block, nil
}
```

SetPreference

`SetPreference` implements the `block.ChainVM`. It sets the preferred block ID.

```
// SetPreference sets the block with ID [ID] as the preferred block
func (vm *VM) SetPreference(id ids.ID) error {
    vm.preferred = id
    return nil
}
```

Other Functions

These functions needs to be implemented for `block.ChainVM`. Most of them are just blank functions returning `nil`.

```
// Bootstrapped marks this VM as bootstrapped
func (vm *VM) Bootstrapped() error { return nil }

// Bootstrapping marks this VM as bootstrapping
func (vm *VM) Bootstrapping() error { return nil }
```

```

// Returns this VM's version
func (vm *VM) Version() (string, error) {
    return Version.String(), nil
}

func (vm *VM) Connected(id ids.ShortID, nodeVersion version.Application) error {
    return nil // noop
}

func (vm *VM) Disconnected(id ids.ShortID) error {
    return nil // noop
}

// This VM doesn't (currently) have any app-specific messages
func (vm *VM) AppGossip(nodeID ids.ShortID, msg []byte) error {
    return nil
}

// This VM doesn't (currently) have any app-specific messages
func (vm *VM) AppRequest(nodeID ids.ShortID, requestID uint32, time time.Time, request []byte) error {
    return nil
}

// This VM doesn't (currently) have any app-specific messages
func (vm *VM) AppResponse(nodeID ids.ShortID, requestID uint32, response []byte) error {
    return nil
}

// This VM doesn't (currently) have any app-specific messages
func (vm *VM) AppRequestFailed(nodeID ids.ShortID, requestID uint32) error {
    return nil
}

// Health implements the common.VM interface
func (vm *VM) HealthCheck() (interface{}, error) { return nil, nil }

```

Factory

VMs should implement the `Factory` interface. `New` method in the interface returns a new VM instance.

```

var _ vms.Factory = &Factory{}

// Factory ...
type Factory struct{}

// New ...
func (f *Factory) New(*snow.Context) (interface{}, error) { return &VM{}, nil }

```

Static API

A VM may have a static API, which allows clients to call methods that do not query or update the state of a particular blockchain, but rather apply to the VM as a whole. This is analogous to static methods in computer programming. AvalancheGo uses [Gorilla's RPC library](#) to implement HTTP APIs.

`StaticService` implements the static API for our VM.

```

// StaticService defines the static API for the timestamp vm
type StaticService struct{}

```

Encode

For each API method, there is:

- A struct that defines the method's arguments
- A struct that defines the method's return values
- A method that implements the API method, and is parameterized on the above 2 structs

This API method encodes a string to its byte representation using a given encoding scheme. It can be used to encode data that is then put in a block and proposed as the next block for this chain.

```

// EncodeArgs are arguments for Encode
type EncodeArgs struct {
    Data      string      `json:"data"`
    Encoding Encoding   `json:"encoding"`
    Length   int32       `json:"length"`
}

// EncodeReply is the reply from Encoder
type EncodeReply struct {

```

```

Bytes      string          `json:"bytes"`
Encoding  formatting.Encoding `json:"encoding"`
}

// Encoder returns the encoded data
func (ss *StaticService) Encode(_ *http.Request, args *EncodeArgs, reply *EncodeReply) error {
    if len(args.Data) == 0 {
        return fmt.Errorf("argument Data cannot be empty")
    }
    var argBytes []byte
    if args.Length > 0 {
        argBytes = make([]byte, args.Length)
        copy(argBytes, args.Data)
    } else {
        argBytes = []byte(args.Data)
    }

    bytes, err := formatting.EncodeWithChecksum(args.Encoding, argBytes)
    if err != nil {
        return fmt.Errorf("couldn't encode data as string: %s", err)
    }
    reply.Bytes = bytes
    reply.Encoding = args.Encoding
    return nil
}

```

Decode

This API method is the inverse of `Encode`.

```

// DecoderArgs are arguments for Decode
type DecoderArgs struct {
    Bytes      string          `json:"bytes"`
    Encoding  formatting.Encoding `json:"encoding"`
}

// DecoderReply is the reply from Decoder
type DecoderReply struct {
    Data      string          `json:"data"`
    Encoding  formatting.Encoding `json:"encoding"`
}

// Decoder returns the Decoded data
func (ss *StaticService) Decode(_ *http.Request, args *DecoderArgs, reply *DecoderReply) error {
    bytes, err := formatting.Decode(args.Encoding, args.Bytes)
    if err != nil {
        return fmt.Errorf("couldn't Decode data as string: %s", err)
    }
    reply.Data = string(bytes)
    reply.Encoding = args.Encoding
    return nil
}

```

API

A VM may also have a non-static HTTP API, which allows clients to query and update the blockchain's state.

Service 's declaration is:

```

// Service is the API service for this VM
type Service struct{ vm *VM }

```

Note that this struct has a reference to the VM, so it can query and update state.

This VM's API has two methods. One allows a client to get a block by its ID. The other allows a client to propose the next block of this blockchain. The blockchain ID in the endpoint changes, since every blockchain has an unique ID.

`timestampvm.getBlock`

Get a block by its ID. If no ID is provided, get the latest block.

`getBlock` Signature

```

timestampvm.getBlock(id: string) ->
{
    id: string,
    data: string,
    timestamp: int,
}

```

```
    parentID: string
}
```

- `id` is the ID of the block being retrieved. If omitted from arguments, gets the latest block
- `data` is the base 58 (with checksum) representation of the block's 32 byte payload
- `timestamp` is the Unix timestamp when this block was created
- `parentID` is the block's parent

getBlock Example Call

```
curl -X POST --data '{  
  "jsonrpc": "2.0",  
  "method": "timestampvm.getBlock",  
  "params":{  
    "id":"xqQV1jDnCXDxhfnNT7tDBcXeoH2jC3Hh7Pyv4GXElz1hfup5K"  
  },  
  "id": 1  
' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/sw813hGSWH8pdU9uzaYy9fCtYffY7AjDd2c9rm64SbApnvjmk
```

getBlock Example Response

```
{  
  "jsonrpc": "2.0",  
  "result": {  
    "timestamp": "1581717416",  
    "data": "11111111111111111111111111111111LpoYY",  
    "id": "xqQV1jDnCXDxhfnNT7tDBcXeoH2jC3Hh7Pyv4GXElz1hfup5K",  
    "parentID": "22XLgiMdfCwTY9iZnVk8ZPuPe3aSrdVr5Dfrbx3ejpJd7oef"  
  },  
  "id": 1  
}
```

getBlock Implementation

```
// GetBlockArgs are the arguments to GetBlock  
type GetBlockArgs struct {  
    // ID of the block we're getting.  
    // If left blank, gets the latest block  
    ID *ids.ID `json:"id"  
}  
  
// GetBlockReply is the reply from GetBlock  
type GetBlockReply struct {  
    Timestamp json.Uint64 `json:"timestamp"` // Timestamp of most recent block  
    Data      string     `json:"data"`      // Data in the most recent block. Base 58 repr. of 5 bytes.  
    ID        ids.ID    `json:"id"`       // String repr. of ID of the most recent block  
    ParentID  ids.ID    `json:"parentID"` // String repr. of ID of the most recent block's parent  
}  
  
// GetBlock gets the block whose ID is [args.ID]  
// If [args.ID] is empty, get the latest block  
func (s *Service) GetBlock(_ *http.Request, args *GetBlockArgs, reply *GetBlockReply) error {  
    // If an ID is given, parse its string representation to an ids.ID  
    // If no ID is given, ID becomes the ID of last accepted block  
    var (  
        id ids.ID  
        err error  
    )  
  
    if args.ID == nil {  
        id, err = s.vm.state.GetLastAccepted()  
        if err != nil {  
            return errCannotGetLastAccepted  
        }  
    } else {  
        id = *args.ID  
    }  
  
    // Get the block from the database  
    block, err := s.vm.getBlock(id)  
    if err != nil {  
        return errNoSuchBlock  
    }  
  
    // Fill out the response with the block's data  
    reply.ID = block.ID()  
    reply.Timestamp = json.Uint64(block.Timestamp().Unix())
```

```

    reply.ParentID = block.Parent()
    data := block.Data()
    reply.Data, err = formatting.EncodeWithChecksum(formatting.CB58, data[:])

    return err
}

```

timestampvm.proposeBlock

Propose the next block on this blockchain.

proposeBlock Signature

```
timestampvm.proposeBlock({data: string}) -> {success: bool}
```

- `data` is the base 58 (with checksum) representation of the proposed block's 32 byte payload.

proposeBlock Example Call

```
curl -X POST --data '{
  "jsonrpc": "2.0",
  "method": "timestampvm.proposeBlock",
  "params": {
    "data": "SkB92YpWm4Q2iPnLGCuDPZPgUQMxajqQQuz91oi3xD984f8r"
  },
  "id": 1
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/bc/sw813hGSWH8pdU9uzaYy9fCtYFFY7AjDd2c9rm64SbApnvjmk
```

PROPOSEBLOCK EXAMPLE RESPONSE

```
{
  "jsonrpc": "2.0",
  "result": {
    "Success": true
  },
  "id": 1
}
```

proposeBlock Implementation

```
// ProposeBlockArgs are the arguments to ProposeValue
type ProposeBlockArgs struct {
    // Data for the new block. Must be base 58 encoding (with checksum) of 32 bytes.
    Data string
}

// ProposeBlockReply is the reply from function ProposeBlock
type ProposeBlockReply struct{
    // True if the operation was successful
    Success bool
}

// ProposeBlock is an API method to propose a new block whose data is [args].Data.
// [args].Data must be a string repr. of a 32 byte array
func (s *Service) ProposeBlock(_ *http.Request, args *ProposeBlockArgs, reply *ProposeBlockReply) error {
    bytes, err := formatting.Decode(formatting.CB58, args.Data)
    if err != nil || len(bytes) != dataLen {
        return errBadData
    }

    var data [dataLen]byte           // The data as an array of bytes
    copy(data[:], bytes[:dataLen]) // Copy the bytes in dataSlice to data

    s.vm.proposeBlock(data)
    reply.Success = true
    return nil
}
```

Plugin

In order to make this VM compatible with `go-plugin`, we need to define a `main` package and method, which serves our VM over gRPC so that AvalancheGo can call its methods.

`main.go`'s contents are:

```
func main() {
    log.Root().SetHandler(log.LvlFilterHandler(log.LvlDebug, log.StreamHandler(os.Stderr, log.TerminalFormat())))
}
```

```

    plugin.Serve(&plugin.ServeConfig{
        HandshakeConfig: rpcchainvm.Handshake,
        Plugins: map[string]plugin.Plugin{
            "vm": rpcchainvm.New(&timestamppvm.VM{}),
        },
        // A non-nil value here enables gRPC serving for this plugin...
        GRPCServer: plugin.DefaultGRPCServer,
    })
}

```

Now AvalancheGo's `rpcchainvm` can connect to this plugin and calls its methods.

Executable Binary

This VM has a [build script](#) that builds an executable of this VM (when invoked, it runs the `main` method from above.)

The path to the executable, as well as its name, can be provided to the build script via arguments. For example:

```
./scripts/build.sh ./avalanchego/build/plugins timestamppvm
```

If no argument is given, the path defaults to a binary named with default VM ID: `$GOPATH/src/github.com/avalanche/avalanchego/build/plugins/tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH`

This name `tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH` is the CB58 encoded 32 byte identifier for the VM. For the timestamppvm, this is the string "timestamppvm" zero-extended in a 32 byte array and encoded in CB58.

VM Aliases

Each VM has a predefined, static ID. For instance, the default ID of the TimestampVM is: `tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH`.

It's possible to give an alias for these IDs. For example, we can alias `TimestampVM` by creating a JSON file at `~/avalanchego/configs/vms/aliases.json` with:

```
{
  "tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH": [
    "timestamppvm",
    "timestamp"
  ]
}
```

Installing a VM

AvalancheGo searches for and registers plugins under the `plugins` [directory](#).

To install the virtual machine onto your node, you need to move the built virtual machine binary under this directory. Virtual machine executable names must be either a full virtual machine ID (encoded in CB58), or a VM alias.

Copy the binary into the `plugins` directory.

```
cp -n <path to your binary> $GOPATH/src/github.com/avalanche/avalanchego/build/plugins/
```

Node Is Not Running

If your node isn't running yet, you can install all virtual machines under your `plugin` directory by starting the node.

Node Is Already Running

Load the binary with the `loadVMs` API.

```
curl -sX POST --data '{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "admin.loadVMs",
  "params": {}
}' -H 'content-type:application/json;' 127.0.0.1:9650/ext/admin
```

Confirm the response of `loadVMs` contains the newly installed virtual machine `tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH`. You'll see this virtual machine as well as any others that weren't already installed previously in the response.

```
{
  "jsonrpc": "2.0",
  "result": {
    "newVMs": [
      "tGas3T58KzdjLHhBDMnH2TvrddhqTji5iZAMZ3RXs2NLpSnhH": [
        "timestamppvm",
        "timestamp"
      ],
    ]
  }
}
```

```

        "spdxUxVJQbx85MGxMhbKw1sHxMnSqJ3QBzDyDYEh6TLuxqQ": []
    },
    "id": 1
}

```

Now, this VM's static API can be accessed at endpoints `/ext/vm/timestampvm` and `/ext/vm/timestamp`. For more details about VM configs, see [here](#).

In this tutorial, we used the VM's ID as the executable name to simplify the process. However, AvalancheGo would also accept `timestampvm` or `timestamp` since those are registered aliases in previous step.

Wrapping Up

That's it! That's the entire implementation of a VM which defines a blockchain-based timestamp server.

In this tutorial, we learned:

- The `block.ChainVM` interface, which all VMs that define a linear chain must implement
- The `snowman.Block` interface, which all blocks that are part of a linear chain must implement
- The `rpccainvm` type, which allows blockchains to run in their own processes.
- An actual implementation of `block.ChainVM` and `snowman.Block`.

Create a Subnet with a Custom Virtual Machine

Avalanche-CLI is capable of doing much than just deploying prebuilt EVM Subnets. This tutorial walks you through the process of creating a Subnet with a custom virtual machine and deploying it locally. Although the tutorial uses a fork of Subnet-EVM as an example, you can extend its lessons to support any custom VM binary.

Fork Subnet-EVM

Instead of building a custom VM from scratch, this tutorial starts with forking Subnet-EVM.

Clone Subnet-EVM

First off, clone the Subnet-EVM repository into a directory of your choosing.

```
git clone https://github.com/ava-labs/subnet-evm.git
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/subnet-evm.git
```

You can find more about SSH and how to use it [here](#). :::

Modify Subnet-EVM

To prove you're running your custom binary and not the stock Subnet-EVM included with Avalanche-CLI, you need to modify the Subnet-EVM binary by making a minor change. Navigate to the directory you cloned Subnet-EVM into and open `./scripts/versions.sh` in a text editor.

The file should look something like this:

```

#!/usr/bin/env bash

# Set up the versions to be used
subnet_evm_version=${SUBNET_EVM_VERSION:-'v0.4.4'}
# Don't export them as they're used in the context of other calls
avalanche_version=${AVALANCHE_VERSION:-'v1.9.3'}
network_runner_version=${NETWORK_RUNNER_VERSION:-'35be10cd3867a94fbe960a1c14a455f179de60d9'}
ginkgo_version=${GINKGO_VERSION:-'v2.2.0'}

# This won't be used, but it's here to make code syncs easier
latest_coreth_version=0.11.3

```

Modify the file by setting a custom version. This tutorial uses `v6.6.6`, but the actual version you choose doesn't matter.

Replace the fourth line in the file to be:

```
subnet_evm_version="v6.6.6"
```

Now build your custom binary by running

```
./scripts/build.sh custom_evm.bin
```

This command builds the binary and saves it at `./custom_vm.bin`.

Create a Custom Genesis

To start a VM, you need to provide a genesis file. Here is a basic Subnet-EVM genesis that's compatible with your custom VM.

Open a text editor and copy the preceding text into a file called `custom_genesis.json`.

Create the Subnet Configuration

Now that you have your binary, it's time to create the Subnet configuration. This tutorial uses `myCustomSubnet` as its Subnet name. Invoke the Subnet Creation Wizard with this command:

```
avalanche subnet create myCustomSubnet
```

Choose Your VM

Select **Custom** for your VM.

Use the arrow keys to navigate: ↓ ↑ → ←
? Choose your VM:
 Subnet-EVM
▶ Custom

Enter the Path to Your Genesis

Enter the path to the genesis file you created in this [step](#).

✓ Enter path to custom genesis: ./custom_genesis.json

Enter the Path to Your VM Binary

Next, enter the path to your VM binary. This should be the path to the `custom_evm.bin` you created previously.

```
✓ Enter path to vm binary: ./custom_vm.bin
```

Wrapping Up

If all worked successfully, the command prints `Successfully created Subnet configuration.`.

Now it's time to deploy it.

Deploy the Subnet Locally

To deploy your Subnet, run

```
avalanche subnet deploy myCustomSubnet
```

Make sure to substitute the name of your Subnet if you used a different one than `myCustomSubnet`.

Next, select Local Network.

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to deploy on:
  ▶ Local Network
    Fuji
    Mainnet
```

This command boots a five node Avalanche network on your machine. It needs to download the latest versions of AvalancheGo and Subnet-EVM. The command may take a couple minutes to run.

If all works as expected, the command output should look something like this:

```
> avalanche subnet deploy myCustomSubnet
✓ Local Network
Deploying [myCustomSubnet] to Local Network
Backend controller started, pid: 95803, output at: /Users/connor/.avalanche-cli/runs/server_20221206_174845/avalanche-clie-
backend
Installing avalanchego-v1.9.3...
avalanchego-v1.9.3 installation successful
VMs ready.
Starting network...
.....
Blockchain has been deployed. Wait until network acknowledges...
.
Network ready to use. Local network node endpoints:
+-----+-----+-----+-----+
| NODE | VM | URL |
+-----+-----+-----+-----+
| node1 | myCustomSubnet | http://127.0.0.1:9650/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc |
+-----+-----+-----+-----+
| node2 | myCustomSubnet | http://127.0.0.1:9652/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc |
+-----+-----+-----+-----+
| node3 | myCustomSubnet | http://127.0.0.1:9654/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc |
+-----+-----+-----+-----+
| node4 | myCustomSubnet | http://127.0.0.1:9656/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc |
+-----+-----+-----+-----+
| node5 | myCustomSubnet | http://127.0.0.1:9658/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc |
+-----+-----+-----+-----+

Browser Extension connection details (any node URL from above works):
RPC URL: http://127.0.0.1:9650/ext/bc/2eVmtiGvE4A9AEPwvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc
```

You can use the `RPC URL` to connect to and interact with your Subnet.

Interact with Your Subnet

Check the Version

You can verify that your Subnet has deployed correctly by querying the local node to see what Subnets it's running. You need to use the [getNodeVersion](#) endpoint. Try running this curl command:

```
curl --location --request POST 'http://127.0.0.1:9650/ext/info' \
--header 'Content-Type: application/json' \
--data='{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "info.getNodeVersion",
  "params": {}'
```

```
}
```

The command returns a list of all the VMs your local node is currently running along with their versions.

```
{
  "jsonrpc": "2.0",
  "result": {
    "version": "avalanche/1.9.3",
    "databaseVersion": "v1.4.5",
    "rpcProtocolVersion": "19",
    "gitCommit": "51c5edd85ccc7927443b945b427e64d91ff99f67",
    "vmVersions": {
      "avm": "v1.9.3",
      "evm": "v0.11.3",
      "platform": "v1.9.3",
      "qDMnZ895HKpRXA2wEvujJew8nNFEkvrH5frCR9T1Suk1sREe": "v6.6.6@e3cbd01f63209b24e8e024e20cb4c17d37f26855"
    }
  },
  "id": 1
}
```

Your results may be slightly different, but you can see that in addition to the X-Chain's `avm`, the C-Chain's `evm`, and the P-Chains' `platform` VM, the node is running the custom VM with version `v6.6.6`.

Check a Balance

If you used the default genesis, your custom VM has a prefunded address. You can verify its balance with a curl command. Make sure to substitute the command's URL with the `RPC URL` from your deployment output.

```
curl --location --request POST 'http://127.0.0.1:9650/ext/bc/2eVmtiGvE4A9AE PvqEWT4reCASPziNZA27goXMRqQ25Y6oaYm/rpc' \
--header 'Content-Type: application/json' \
--data=raw '{
  "jsonrpc": "2.0",
  "method": "eth_getBalance",
  "params": [
    "0x8db97c7cece249c2b98bcd0226cc4c2a57bf52fc",
    "latest"
  ],
  "id": 1
}'
```

The command should return

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0xd3c21bcecceda1000000"
}
```

The balance is hex encoded, so this means the address has a balance of 1 million tokens.

Note, this command doesn't work on all custom VMs, only VMs that implement the EVM's `eth_getBalance` interface.

Next Steps

You've now unlocked the ability to deploy custom VMs. Go build something cool!

How to Create an EVM-Based Subnet Configuration

To create an Ethereum Virtual Machine (EVM) compatible Subnet, Avalanche-CLI runs [Subnet-EVM](#) as its virtual machine.

Prerequisites

- [Avalanche-CLI installed](#)

Using the Subnet Creation Wizard

The Subnet Creation Wizard walks you through the process of creating your Subnet. To get started, pick a name for your Subnet and run

```
avalanche subnet create <subnetName>
```

The following sections walk through each question in the wizard.

Choose Your VM

Select `SubnetEVM`.

Enter Your Subnet's ChainID

Choose a unique positive integer for your EVM-style ChainID. Visit [chainlist](#) to verify that your selection is indeed unique.

Token Symbol

Enter a string to name your Subnet's native token. The token symbol doesn't necessarily need to be unique. Example token symbols are AVAX, JOE, and BTC.

Subnet-EVM Version

Select `Use latest version`.

Gas Fee Configuration

Select your fee configuration. The Ava Labs team highly recommends `Low disk use / Low Throughput 1.5 mil gas/s (C-Chain's setting)`.

Airdrop

For development Subnets, select `Airdrop 1 million tokens to the default address (do not use in production)`.

When you are ready to start more mature testing, select `Customize your airdrop` to distribute funds to additional addresses.

Precompiles

If you'd like to add precompiles to customize your Subnet, select `Yes`.

If you don't or don't know what that means, select `No`.

Wrapping Up

If the command works successfully, it prints `Successfully created subnet configuration`.

Using a Custom Genesis

The Subnet Creation Wizard won't customize every aspect of the Subnet-EVM genesis for you. If you'd like complete control, you can specify a custom genesis by providing a path to the file you'd like to use. This bypasses most of the wizard's prompts. Run with:

```
avalanche subnet create <subnetName> --genesis <filepath>
```

Overwriting an Existing Subnet Config

By default, creating a Subnet configuration with the same `subnetName` as one that already exists fails. To overwrite an existing config, use the `-f` force flag:

```
avalanche subnet create <existingSubnetName> -f
```

Cross-Subnet Communication

Avalanche Warp Messaging (AWM) enables native cross-Subnet communication and allows [Virtual Machine \(VM\)](#) developers to implement arbitrary communication protocols between any two Subnets.

Use Cases

Use cases for AWM may include but is not limited to:

- Oracle Networks: Connecting a Subnet to an oracle network is a costly process. AWM makes it easy for oracle networks to broadcast their data from their origin chain to other Subnets.
- Token transfers between Subnets
- State Sharding between multiple Subnets

Elements of Cross-Subnet Communication

The communication consists of the following four steps:



Signing Messages on the Origin Subnet

AWM is a low-level messaging protocol. Any type of data encoded in an array of bytes can be included in the message sent to another Subnet. AWM uses the [BLS signature scheme](#), which allows message recipients to verify the authenticity of these messages. Therefore, every validator on the Avalanche network holds a BLS key pair, consisting of a private key for signing messages and a public key that others can use to verify the signature.

Signature Aggregation on the Origin Subnet

If the validator set of a Subnet is very large, this would result in the Subnet's validators sending many signatures between them. One of the powerful features of BLS is the ability to aggregate many signatures of different signers in a single multi-signature. Therefore, validators of one Subnet can now individually sign a message and these signatures are then aggregated into a short multi-signature that can be quickly verified.

Delivery of Messages to the Destination Subnet

The messages do not pass through a central protocol or trusted entity, and there is no record of messages sent between Subnets on the primary network. This avoids a bottleneck in Subnet-to-Subnet communication, and non-public Subnets can communicate privately.

It is up to the Subnets and their users to determine how they want to transport data from the validators of the origin Subnet to the validators of the destination Subnet and what guarantees they want to provide for the transport.

Verification of Messages in the Destination Subnet

When a Subnet wants to process another Subnet's message, it will look up both BLS Public Keys and stake of the origin Subnet. The authenticity of the message can be verified using these public keys and the signature.

The combined weight of the validators that must be part of the BLS multi-signature to be considered valid can be set according to the individual requirements of each Subnet-to-Subnet communication. Subnet A may accept messages from Subnet B that are signed by at least 70% of stake. Messages from Subnet C are only accepted if they have been signed by validators that account for 90% of the stake.

Since all validators' public keys of the validators and their stake weights are recorded on the primary network's P-chain, they are readily accessible to any virtual machine run by the validators. Therefore, the Subnets do not need to communicate with each other about changes in their respective sets of validators, but can simply rely on the latest information on the P-Chain. Therefore, AWM introduces no additional trust assumption other than that the validators of the origin Subnet are participating honestly.

Reference Implementation

We created a Proof-of-Concept VM called [XSVM](#). This VM allows for simple AWM transfers between any two Subnets running it out-of-the-box.

Description: How to customize a Subnet by utilizing Genesis, Precompile and Blockchain Configs.

Customize Your EVM-Powered Subnet

All Subnets can be customized by utilizing [Subnet Configs](#).

A Subnet can have one or more blockchains. For example, the Primary Network, which is a Subnet, a special one nonetheless, has 3 blockchains. Each chain can be further customized using chain specific configuration file. See [here](#) for detailed explanation.

A blockchain created by or forked from [Subnet-EVM](#) can be customized by utilizing one or more of the following methods:

- [Genesis](#)
- [Precompile](#)
- [Upgrade Configs](#)
- [Chain Configs](#)

Subnet Configs

A Subnet can be customized by setting parameters for the following:

- [Validator-only communication to create a private Subnet](#)
- [Consensus](#)
- [Gossip](#)

See [here](#) for more info.

Genesis

Each blockchain has some genesis state when it's created. Each Virtual Machine defines the format and semantics of its genesis data.

The default genesis Subnet-EVM provided below has some well defined parameters:

```
{  
  "config": {  
    "chainId": 43214,  
    "homesteadBlock": 0,  
    "eip150Block": 0,  
    "eip150Hash": "0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",  
    "eip155Block": 0,  
    "eip158Block": 0,  
    "byzantiumBlock": 0,  
    "constantinopleBlock": 0,  
    "petersburgBlock": 0,  
    "istanbulBlock": 0,  
    "muirGlacierBlock": 0,  
    "subnetEVMTimestamp": 0,  
    "feeConfig": {  
      "gasLimit": 15000000,  
      "minBaseFee": 25000000000,  
      "targetGas": 15000000,  
      "baseFeeChangeDenominator": 36,  
      "minBlockGasCost": 0,  
      "maxBlockGasCost": 1000000,  
      "maxFeeReciprocity": 0.01  
    }  
  }  
}
```

Chain Config

`chainID` : Denotes the ChainID of to be created chain. Must be picked carefully since a conflict with other chains can cause issues. One suggestion is to check with [chainlist.org](#) to avoid ID collision, reserve and publish your ChainID properly.

You can use `eth getChainConfig` RPC call to get the current chain config. See [here](#) for more info.

Hard Forks

`homesteadBlock`, `eip150Block`, `eip150Hash`, `eip155Block`, `byzantiumBlock`, `constantinopleBlock`, `petersburgBlock`, `istanbulBlock`, `muirGlacierBlock`. `subnetEVMTimestamp` are hard fork activation times. Changing these may cause issues, so treat them carefully.

Fee Config

`gasLimit` : Sets the max amount of gas consumed per block. This restriction puts a cap on the amount of computation that can be done in a single block, which in turn sets a limit on the maximum gas usage allowed for a single transaction. For reference, C-Chain value is set to `15,000,000`.

`targetBlockRate` : Sets the target rate of block production in seconds. A target of 2 will target producing a block every 2 seconds. If the network starts producing blocks at a faster rate, it indicates that more blocks than anticipated are being issued to the network, resulting in an increase in base fees. For C-chain this value is set to 2.

`minBaseFee` : Sets a lower bound on the EIP-1559 base fee of a block. Since the block's base fee sets the minimum gas price for any transaction included in that block, this effectively sets a minimum gas price for any transaction.

`targetGas` : Specifies the targeted amount of gas (including block gas cost) to consume within a rolling 10-seconds window. When the dynamic fee algorithm observes that network activity is above/below the `targetGas`, it increases/decreases the base fee proportionally to how far above/below the target actual network activity is. If the network starts producing blocks with gas cost higher than this base fee, an increase accordingly.

`baseFeeChangeDenominator` : Divides the difference between actual and target utilization to determine how much to increase/decrease the base fee. A larger denominator indicates a slower changing, stickier base fee, while a lower denominator allows the base fee to adjust more quickly. For reference, the C-chain value is set to `25`. This value sets the base fee to increase or decrease by a factor of `1/25` of the parent block's base fee.

If it is produced faster/slower, the block gas cost will be increased/decreased by the step value for each second faster/slower than the target block rate.

Custom Fee Recipients

Section 3: Writing Good Algorithms

The fields `nonce`, `timestamp`, `extraData`, `gasLimit`, `difficulty`, `mixHash`, `coinbase`, `number`, `gasUsed`, `parentHash` defines the genesis block. The fields `receiptsRoot` and `stateRoot` define the root hash of the state tree. `Logs` is a list of logs of the three contracts.

fields.

`nonce`, `mixHash` and `difficulty` are remnant parameters from Proof of Work systems. For Avalanche, these don't play any relevant role, so you should just leave them as their default values:

nonce : The result of the mining process iteration is this value. It can be any value in the genesis block. Default value is 0x0 .

Digitized by srujanika@gmail.com

difficulty : The difficulty level applied during the nonce discovering process of this block. Default value is 0x0

`timestamp` : The timestamp of the creation of the genesis block. This is commonly set to 0x0.

`extraData` : Optional extra data that can be included in the genesis block. This is commonly set to `0x`

`gasLimit` : The total amount of gas that can be used in a single block. It should be set to the same value as in the [fee config](#). The value `e4e1c0` is hexadecimal and is equal to `15,000,000`.

`coinbase` : Refers to the address of the block producers. This also means it represents the recipient of the block reward. It is usually set to `0x00` for the genesis block. To allow fee recipients in Subnet-EVM, refer to [this section](#).

`gasUsed` : This is the amount of gas used by the genesis block. It is usually set to `0x0`.

number : This is the number of the genesis block. This required to be `0x0` for the genesis. Otherwise it will error.

Genesis Examples

Another example of a genesis file can be found in the [networks folder](#). Note: please remove `airdropHash` and `airdropAmount` fields if you want to start with it.

Here are a few examples on how a genesis file is used:

- scripts/run.sh

Setting the Genesis Allocation

Alloc defines addresses and their initial balances. This should be changed accordingly for each chain. If you don't provide any genesis allocation, you won't be able to interact with your new chain (all transactions require a fee to be paid from the sender's balance).

```
"alloc": {
    "8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC": {
        "balance": "0x295BE96E64066972000000"
    }
}
```

To specify a different genesis allocation, populate the `alloc` field in the genesis JSON as follows:

```
        "alloc": {
            "8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC": {
                "balance": "0x52B7D2DCC80CD2E4000000"
            },
            "Ab5801a7D398351b8bE11C439e05C5B3259aeC9B": {
                "balance": "0xa796504b1cb5a7c0000"
            }
        },
    }
```

The keys in the allocation are [hex](#) addresses without the canonical `0x` prefix. The balances are denominated in Wei ([10^18 Wei = 1 Whole Unit of Native Token](#)) and expressed as hex strings with the canonical `0x` prefix. You can use [this converter](#) to translate between decimal and hex numbers.

The above example yields the following genesis allocations (denominated in whole units of the native token, that is 1 AVAX/1 WAGMI):

0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52Fc: 100000000 (0x52B7D2DCC80CD2E400000=10000000000000000000000000000000 Wei)
0xAb5801a7D398351b8bE11C439e05C5B3259aeC9B: 49463 (0xa796504b1cb5a7c0000=49463000000000000000000 Wei)

Setting a Custom Fee Recipient

By default, all fees are burned (sent to the black hole address with "allowFeeRecipients": false). However, it is possible to enable block producers to set a fee recipient (who will get compensated for blocks they produce).

To enable this feature, you'll need to add the following to your genesis file (under the "config" key):

```
{  
  "config": {
```

```

        "allowFeeRecipients": true
    }
}

```

Fee Recipient Address

With `allowFeeRecipients` enabled, your validators can specify their addresses to collect fees. They need to update their EVM [chain config](#) with the following to specify where the fee should be sent to.

```

{
  "feeRecipient": "<YOUR 0x-ADDRESS>"
}

```

:::warning

If `allowFeeRecipients` feature is enabled on the Subnet, but a validator doesn't specify a "feeRecipient", the fees will be burned in blocks it produces.

:::

Note: this mechanism can be also activated as a precompile. See [Changing Fee Reward Mechanisms](#) section for more details.

Precompiles

Subnet-EVM can provide custom functionalities with precompiled contracts. These precompiled contracts can be activated through `ChainConfig` (in genesis or as an upgrade).

AllowList Interface

The `AllowList` interface is used by precompiles to check if a given address is allowed to use a precompiled contract. `AllowList` consist of two main roles, `Admin` and `Enabled`. `Admin` can add/remove other `Admin` and `Enabled` addresses. `Enabled` addresses can use the precompiled contract, but cannot modify other roles.

`AllowList` adds `adminAddresses` and `enabledAddresses` fields to precompile contract configurations. For instance fee manager precompile contract configuration looks like this:

```

{
  "feeManagerConfig": {
    "blockTimestamp": 0,
    "adminAddresses": [<list of addresses>],
    "enabledAddresses": [<list of addresses>]
  }
}

```

`AllowList` configuration affects only the related precompile. For instance, the admin address in `feeManagerConfig` does not affect admin addresses in other activated precompiles.

The `AllowList` solidity interface is defined as follows, and can be found in [IAllowList.sol](#):

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IAllowList {
    // Set [addr] to have the admin role over the precompile
    function setAdmin(address addr) external;

    // Set [addr] to be enabled on the precompile contract.
    function setEnabled(address addr) external;

    // Set [addr] to have no role the precompile contract.
    function setNone(address addr) external;

    // Read the status of [addr].
    function readAllowList(address addr) external view returns (uint256 role);
}

```

`readAllowList(addr)` will return a `uint256` with a value of 0, 1, or 2, corresponding to the roles `None`, `Enabled`, and `Admin` respectively.

Note: `AllowList` is not an actual contract but just an interface. It's not callable by itself. This is used by other precompiles. Check other precompile sections to see how this works.

Restricting Smart Contract Deployers

If you'd like to restrict who has the ability to deploy contracts on your Subnet, you can provide an `AllowList` configuration in your genesis or upgrade file:

```

{
  "contractDeployerAllowListConfig": {
    "blockTimestamp": 0,
    "adminAddresses": ["0x8db97c7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
}

```

```
}
```

In this example, `0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC` is named as the `Admin` of the `ContractDeployerAllowList`. This enables it to add other `Admin` or to add `Enabled` addresses. Both `Admin` and `Enabled` can deploy contracts. To provide a great UX with factory contracts, the `tx.Origin` is checked for being a valid deployer instead of the caller of `CREATE`. This means that factory contracts will still be able to create new contracts as long as the sender of the original transaction is an allow listed deployer.

The `Stateful Precompile` contract powering the `ContractDeployerAllowList` adheres to the [AllowList Solidity interface](#) at `0x0200` (you can load this interface and interact directly in Remix):

- If you attempt to add a `Enabled` and you are not an `Admin`, you will see something like: 
- If you attempt to deploy a contract but you are not an `Admin` not a `Enabled`, you will see something like: 
- If you call `readAllowList(addr)` then you can read the current role of `addr`, which will return a `uint256` with a value of 0, 1, or 2, corresponding to the roles `None`, `Enabled`, and `Admin` respectively.

:::warning

If you remove all of the admins from the allow list, it will no longer be possible to update the allow list without modifying the Subnet-EVM to schedule a network upgrade.

:::

Initial Contract Allow List Configuration

It's possible to enable this precompile with an initial configuration to activate its effect on activation timestamp. This provides a way to enable the precompile without an admin address to manage the deployer list. With this, you can define a list of addresses that are allowed to deploy contracts. Since there will be no admin address to manage the deployer list, it can only be modified through a network upgrade. To use initial configuration, you need to specify addresses in `enabledAddresses` field in your genesis or upgrade file:

```
{
  "contractDeployerAllowListConfig": {
    "blockTimestamp": 0,
    "enabledAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
  }
}
```

This will allow only `0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC` to deploy contracts. For further information about precompile initial configurations see [Initial Precompile Configurations](#).

Restricting Who Can Submit Transactions

Similar to restricting contract deployers, this precompile restricts which addresses may submit transactions on chain. Like the previous section, you can activate the precompile by including an `AllowList` configuration in your genesis file:

```
{
  "config": {
    "txAllowListConfig": {
      "blockTimestamp": 0,
      "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
    }
  }
}
```

In this example, `0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC` is named as the `Admin` of the `TransactionAllowList`. This enables them to add other `Admins` or to add `Allowed`. Both `Admins` and `Enabled` can submit transactions to the chain.

The `Stateful Precompile` contract powering the `TxAllowList` adheres to the [AllowList Solidity interface](#) at `0x0200` (you can load this interface and interact directly in Remix):

- If you attempt to add an `Enabled` and you are not an `Admin`, you will see something like: 
- If you attempt to submit a transaction but you are not an `Admin` or not `Enabled`, you will see something like: `cannot issue transaction from non-allow listed address`
- If you call `readAllowList(addr)` then you can read the current role of `addr`, which will return a `uint256` with a value of 0, 1, or 2, corresponding to the roles `None`, `Allowed`, and `Admin` respectively.

:::warning

If you remove all of the admins from the allow list, it will no longer be possible to update the allow list without modifying the Subnet-EVM to schedule a network upgrade.

•

Initial TX Allow List Configuration

It's possible to enable this precompile with an initial configuration to activate its effect on activation timestamp. This provides a way to enable the precompile without an admin address to manage the TX allow list. With this, you can define a list of addresses that are allowed to submit transactions. Since there will be no admin address to manage the TX list, it can only be modified through a network upgrade. To use initial configuration, you need to specify addresses in `enabledAddresses` field in your genesis or upgrade file:

```
{
  "txAllowListConfig": {
    "blockTimestamp": 0,
    "enabledAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
  }
}
```

This will allow only `0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC` to submit transactions. For further information about precompile initial configurations see [Initial Precompile Configurations](#).

Minting Native Coins

You can mint native(gas) coins with a precompiled contract. In order to activate this feature, you can provide `nativeMinterConfig` in genesis:

```
{
  "config": {
    "contractNativeMinterConfig": {
      "blockTimestamp": 0,
      "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
    }
  }
}
```

`adminAddresses` denotes admin accounts who can add other `Admin` or `Enabled` accounts. `Admin` and `Enabled` are both eligible to mint native coins for other addresses. `ContractNativeMinter` uses same methods as in `ContractDeployerAllowList`.

```
// (c) 2022-2023, Ava Labs, Inc. All rights reserved.  
// See the file LICENSE for licensing terms.  
  
pragma solidity ^0.8.0;  
import "./IAccessControl.sol";  
  
interface INativeMinter is IAllowList {  
    // Mint [amount] number of native coins and send to [addr]  
    function mintNativeCoin(address addr, uint256 amount)  
}
```

`mintNativeCoin` takes an address and amount of native coins to be minted. The amount denotes the amount in minimum denomination of native coins (10^{18}). For example, if you want to mint 1 native coin (in AVAX), you need to pass $1 * 10^{18}$ as the amount.

`setEnabled`, `setNone`. For more information see [AllowList Solidity interface](#).

Warning

EVM does not prevent overflows when storing the address balance. Overflows in balance opcodes are handled by setting the balance to maximum. However the same won't apply for API calls. If you try to mint more than the maximum balance, API calls will return the overflowed hex-balance. This can break external tooling. Make sure the total supply of native coins is always less than $2^{256}-1$.

* * *

Initial Native Minter Configuration

It's possible to enable this precompile with an initial configuration to activate its effect on activation timestamp. This provides a way to enable the precompile without an admin address to mint native coins. With this, you can define a list of addresses that will receive an initial mint of the native coin when this precompile activates. This can be useful for networks that require a one-time mint without specifying any admin addresses. To use initial configuration, you need to specify a map of addresses with their corresponding mint amounts in `initialMint` field in your genesis or upgrade file:

```
{  
  "contractNativeMinterConfig": {  
    "blockTimestamp": 0,  
    "initialMint": {  
      "0x8db97C7c1cE249c2b98bDC0226Cc42A57BF52Fc": "10000000000000000000000000000000",  
      "0x10037Fb06Ec4aB8c870a92AE3f00cD58e5D484b3": "0xde0B6b3a7640000"  
    }  
  }  
}
```

```
}
```

In the amount field you can specify either decimal or hex string. This will mint 1000000000000000000 (equivalent of 1 Native Coin denominated as 10^{18}) to both addresses. Note that these are both in string format. "0xde0b6b3a7640000" hex is equivalent to 1000000000000000000. For further information about precompile initial configurations see [Initial Precompile Configurations](#).

Configuring Dynamic Fees

You can configure the parameters of the dynamic fee algorithm on chain using the `FeeConfigManager`. In order to activate this feature, you will need to provide the `FeeConfigManager` in the genesis:

```
{
  "config": {
    "feeManagerConfig": {
      "blockTimestamp": 0,
      "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
    }
  }
}
```

The precompile implements the `FeeManager` interface which includes the same `AllowList` interface used by `ContractNativeMinter`, `TxA.AllowList`, etc. For an example of the `AllowList` interface, see the [TxAllowList](#) above.

The `Stateful Precompile` contract powering the `FeeConfigManager` adheres to the following Solidity interface at `0x020003` (you can load this interface and interact directly in Remix). It can be also found in [IFeeManager.sol](#):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "./IA.AllowList.sol";

interface IFeeManager is IAllowList {
  // Set fee config fields to contract storage
  function setFeeConfig(
    uint256 gasLimit,
    uint256 targetBlockRate,
    uint256 minBaseFee,
    uint256 targetGas,
    uint256 baseFeeChangeDenominator,
    uint256 minBlockGasCost,
    uint256 maxBlockGasCost,
    uint256 blockGasCostStep
  ) external;

  // Get fee config from the contract storage
  function getFeeConfig()
    external
    view
    returns (
      uint256 gasLimit,
      uint256 targetBlockRate,
      uint256 minBaseFee,
      uint256 targetGas,
      uint256 baseFeeChangeDenominator,
      uint256 minBlockGasCost,
      uint256 maxBlockGasCost,
      uint256 blockGasCostStep
    );

  // Get the last block number changed the fee config from the contract storage
  function getFeeConfigLastChangedAt() external view returns (uint256 blockNumber);
}
```

`FeeConfigManager` precompile uses `IAllowList` interface directly, meaning that it uses the same `AllowList` interface functions like `readAllowList` and `setAdmin`, `setEnabled`, `setNone`. For more information see [AllowList Solidity interface](#).

In addition to the `AllowList` interface, the `FeeConfigManager` adds the following capabilities:

- `getFeeConfig` - retrieves the current dynamic fee config
- `getFeeConfigLastChangedAt` - retrieves the timestamp of the last block where the fee config was updated
- `setFeeConfig` - sets the dynamic fee config on chain (see [here](#) for details on the fee config parameters)

You can also get the fee configuration at a block with the `eth_feeConfig` RPC method. For more information see [here](#).

Initial Fee Config Configuration

It's possible to enable this precompile with an initial configuration to activate its effect on activation timestamp. This provides a way to define your fee structure to take effect at the activation. To use the initial configuration, you need to specify the fee config in `initialFeeConfig` field in your genesis or upgrade file:

```
{  
    "feeManagerConfig": {  
        "blockTimestamp": 0,  
        "initialFeeConfig": {  
            "gasLimit": 20000000,  
            "targetBlockRate": 2,  
            "minBaseFee": 100000000,  
            "targetGas": 10000000,  
            "baseFeeChangeDenominator": 48,  
            "minBlockGasCost": 0,  
            "maxBlockGasCost": 10000000,  
            "blockGasCostStep": 500000  
        }  
    }  
}
```

This will set the fee config to the values specified in the `initialFeeConfig` field. For further information about precompile initial configurations see [Initial Precompile Configurations](#).

Changing Fee Reward Mechanisms

Fee reward mechanism can be configured with this stateful precompile contract called as `RewardManager`. Configuration can include burning fees, sending fees to a predefined address, or enabling fees to be collected by block producers. This precompile can be configured as follows in the genesis file:

```
{  
  "config": {  
    "rewardManagerConfig": {  
      "blockTimestamp": 0,  
      "adminAddresses": ["0x8db97C7EcE249c2b98bDC0226Cc4C2A57BF52FC"]  
    }  
  }  
}
```

`adminAddresses` denotes admin accounts who can add other `Admin` or `Enabled` accounts. `Admin` and `Enabled` are both eligible to change the current fee mechanism.

The precompile implements the `RewardManager` interface which includes the `AllowList` interface. For an example of the `AllowList` interface, see the [TxAllowList](#) above.

The Stateful Precompile contract powering the RewardManager adheres to the following Solidity interface at [0x020004](#) (you can load this interface and interact directly in Remix). It can be also found in [RewardManager.sol](#):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
import "./IAccountList.sol";

interface IRewardManager is IAllowList {
    // setRewardAddress sets the reward address to the given address
    function setRewardAddress(address addr) external;

    // allowFeeRecipients allows block builders to claim fees
    function allowFeeRecipients() external;

    // disableRewards disables block rewards and starts burning fees
    function disableRewards() external;

    // currentRewardAddress returns the current reward address
    function currentRewardAddress() external view returns (address rewardAddress);

    // areFeeRecipientsAllowed returns true if fee recipients are allowed
    function areFeeRecipientsAllowed() external view returns (bool isAllowed);
}
```

RewardManager precompile uses `IAllowList` interface directly, meaning that it uses the same `AllowList` interface functions like `readAllowList` and `setAdmin`, `setEnabled`, `setNone`. For more information see [AllowList Solidity interface](#).

In addition to the `AllowList` interface, the `RewardManager` adds the following capabilities:

- `setRewardAddress` - sets the address to which fees are sent. This address can be a contract or a user address. The address becomes the required coinbase address for the blocks that this mechanism is enabled on. Meaning that it will receive the fees collected from the transactions in the block. Receiving fees will not call any contract functions or fallback functions. It will simply increase the balance of the address by the amount of fees.
 - `allowFeeRecipients` - enables block producers to claim fees. This will allow block producers to claim fees by specifying their own addresses in their chain config. See [here](#) for more information on how to specify the fee recipient address in the chain config.
 - `disableRewards` - disables block rewards and starts burning fees.

- `currentRewardAddress` - returns the current reward address. This is the address to which fees are sent. It can include black hole address (`0x010...0`) which means that fees are burned. It can also include a predefined hash (`0x00`) denoting custom fee recipients are allowed. It's advised to use the `areFeeRecipientsAllowed` function to check if custom fee recipients are allowed first.
- `areFeeRecipientsAllowed` - returns true if custom fee recipients are allowed.

These 3 mechanisms (burning, sending to a predefined address, and enabling fees to be collected by block producers) cannot be enabled at the same time. Enabling one mechanism will take over the previous mechanism. For example, if you enable `allowFeeRecipients` and then enable `disableRewards`, the `disableRewards` will take over and fees will be burned.

Note: Reward addresses or fee recipient addresses are not required to be an admin or enabled account.

Initial Configuration

It's possible to enable this precompile with an initial configuration to activate its effect on activation timestamp. This provides a way to enable the precompile without an admin address to change the fee reward mechanism. This can be useful for networks that require a one-time reward mechanism change without specifying any admin addresses. Without this initial configuration, the precompile will inherit the `feeRecipients` mechanism activated at genesis. Meaning that if `allowFeeRecipients` is set to true in the genesis file, the precompile will be enabled with the `allowFeeRecipients` mechanism. Otherwise it will keep burning fees. To use the initial configuration, you need to specify the initial reward mechanism in `initialRewardConfig` field in your genesis or upgrade file.

In order to allow custom fee recipients, you need to specify the `allowFeeRecipients` field in the `initialRewardConfig`:

```
{
  "rewardManagerConfig": {
    "blockTimestamp": 0,
    "initialRewardConfig": {
      "allowFeeRecipients": true
    }
  }
}
```

In order to set an address to receive all transaction rewards, you need to specify the `rewardAddress` field in the `initialRewardConfig`:

```
{
  "rewardManagerConfig": {
    "blockTimestamp": 0,
    "initialRewardConfig": {
      "rewardAddress": "0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"
    }
  }
}
```

In order to disable rewards and start burning fees, you need to leave all fields in the `initialRewardConfig` empty:

```
{
  "rewardManagerConfig": {
    "blockTimestamp": 0,
    "initialRewardConfig": {}
  }
}
```

However this is different than the default behavior of the precompile. If you don't specify the `initialRewardConfig` field, the precompile will inherit the `feeRecipients` mechanism activated at genesis. Meaning that if `allowFeeRecipients` is set to true in the genesis file, the precompile will be enabled with the `allowFeeRecipients` mechanism. Otherwise it will keep burning fees. Example configuration for this case:

```
{
  "rewardManagerConfig": {
    "blockTimestamp": 0,
    "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
  }
}
```

If `allowFeeRecipients` and `rewardAddress` are both specified in the `initialRewardConfig` field then an error will be returned and precompile won't be activated. For further information about precompile initial configurations see [Initial Precompile Configurations](#).

Contract Examples

Subnet-EVM contains example contracts for precompiles under `/contract-examples`. It's a hardhat project with tests, tasks. For more information see [contract examples README](#).

Network Upgrades: Enable/Disable Precompiles

:::warning

Performing a network upgrade requires coordinating the upgrade network-wide. A network upgrade changes the rule set used to process and verify blocks, such that any node that upgrades incorrectly or fails to upgrade by the time that upgrade goes into effect may become out of sync with the rest of the network.

Any mistakes in configuring network upgrades or coordinating them on validators may cause the network to halt and recovering may be difficult.

:::

In addition to specifying the configuration for each of the above precompiles in the genesis chain config, they can be individually enabled or disabled at a given timestamp as a network upgrade. Disabling a precompile disables calling the precompile and destructs its storage so it can be enabled at a later timestamp with a new configuration if desired.

These upgrades must be specified in a file named `upgrade.json` placed in the same directory where `config.json` resides: `{chain-config-dir}/{blockchainID}/upgrade.json`. For example, WAGMI Subnet upgrade should be placed in `~/.avalanchego/configs/chains/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQtTFhBq8qehttt/upgrade.json`.

The content of the `upgrade.json` should be formatted according to the following:

```
{
  "precompileUpgrades": [
    {
      "[PRECOMPILE_NAME)": {
        "blockTimestamp": "[ACTIVATION_TIMESTAMP]", // unix timestamp precompile should activate at
        "[PARAMETER)": "[VALUE]" // precompile specific configuration options, eg. "adminAddresses"
      }
    }
  ]
}
```

:::warning

An invalid `blockTimestamp` in an upgrade file results the update failing. The `blockTimestamp` value should be set to a valid Unix timestamp value which is in the *future* relative to the *head of the chain*. If the node encounters a `blockTimestamp` which is in the past, it will fail on startup.

:::

To disable a precompile, the following format should be used:

```
{
  "precompileUpgrades": [
    {
      "<precompileName>": {
        "blockTimestamp": "[DEACTIVATION_TIMESTAMP]", // unix timestamp the precompile should deactivate at
        "disable": true
      }
    }
  ]
}
```

Each item in `precompileUpgrades` must specify exactly one precompile to enable or disable and the block timestamps must be in increasing order. Once an upgrade has been activated (a block after the specified timestamp has been accepted), it must always be present in `upgrade.json` exactly as it was configured at the time of activation (otherwise the node will refuse to start).

Enabling and disabling a precompile is a network upgrade and should always be done with caution.

:::danger

For safety, you should always treat `precompileUpgrades` as append-only.

As a last resort measure, it is possible to abort or reconfigure a precompile upgrade that has not been activated since the chain is still processing blocks using the prior rule set.

:::

If aborting an upgrade becomes necessary, you can remove the precompile upgrade from `upgrade.json` from the end of the list of upgrades. As long as the blockchain has not accepted a block with a timestamp past that upgrade's timestamp, it will abort the upgrade for that node.

Example

```
{
  "precompileUpgrades": [
    {
      "feeManagerConfig": {
        "blockTimestamp": 1668950000,
        "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
      }
    },
    {
      "txAllowListConfig": {
        "blockTimestamp": 1668960000,
        "adminAddresses": ["0x8db97C7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
      }
    }
  ]
}
```

```

        }
    },
    {
        "feeManagerConfig": {
            "blockTimestamp": 1668970000,
            "disable": true
        }
    }
]
}

```

This example enables the `feeManagerConfig` at the first block with `timestamp >= 1668950000`, enables `txAllowListConfig` at the first block with `timestamp >= 1668960000`, and disables `feeManagerConfig` at the first block with `timestamp >= 1668970000`.

When a precompile disable takes effect (that is, after its `blockTimestamp` has passed), its storage will be wiped. If you want to reenable it, you will need to treat it as a new configuration.

After you have created the `upgrade.json` and placed it in the chain config directory, you need to restart the node for the upgrade file to be loaded (again, make sure you don't restart all Subnet validators at once!). On node restart, it will print out the configuration of the chain, where you can double-check that the upgrade has loaded correctly. In our example:

```

INFO [08-15|15:09:36.772] <2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qegeh6tt Chain>
github.com/ava-labs/subnet-evm/eth/backend.go:155: Initialised chain configuration
config="(ChainID: 11111 Homestead: 0 EIP150: 0 EIP155: 0 EIP158: 0 Byzantium: 0
Constantinople: 0 Petersburg: 0 Istanbul: 0, Muir Glacier: 0, Subnet EVM: 0, FeeConfig:
{\\"gasLimit\":20000000,\\"targetBlockRate\":2,\\"minBaseFee\":\"1000000000,\\"targetGas\
\":100000000,\\"baseFeeChangeDenominator\":48,\\"minBlockGasCost\":0,\\"maxBlockGasCost\
\":10000000,\\"blockGasCostStep\":500000}, AllowFeeRecipients: false, NetworkUpgrades: \
{\\"subnetEVMTimestamp\":0}, PrecompileUpgrade: {}, UpgradeConfig: {\\"precompileUpgrades\":[{\\"feeManagerConfig\":\
{\\"adminAddresses\": [\"0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc\"],\"enabledAddresses\":null,\\"blockTimestamp\":1668950000}},\
{\\"txAllowListConfig\":[\"adminAddresses\": [\"0x8db97c7cece249c2b98bdc0226cc4c2a57bf52fc\"],\"enabledAddresses\":null,\\"blockTimestamp\":1668960000}},\
{\\"feeManagerConfig\":[{\"adminAddresses\":null,\"enabledAddresses\":null,\\"blockTimestamp\":1668970000,\\"disable\":true}]}]}, Engine: Dummy Consensus Engine}"

```

Notice that `precompileUpgrades` entry correctly reflects the changes. You can also check the activated precompiles at a timestamp with the [eth_getActivePrecompilesAt](#) RPC method. The [eth_getChainConfig](#) RPC method will also return the configured upgrades in the response.

That's it; your Subnet is all set and the desired upgrades will be activated at the indicated timestamp!

Initial Precompile Configurations

Precompiles can be managed by some privileged addresses to change their configurations and activate their effects. For example, the `feeManagerConfig` precompile can have `adminAddresses` which can change the fee structure of the network.

```

{
    "precompileUpgrades": [
        {
            "feeManagerConfig": {
                "blockTimestamp": 1668950000,
                "adminAddresses": ["0x8db97c7cEcE249c2b98bDC0226Cc4C2A57BF52FC"]
            }
        }
    ]
}

```

In this example, only the address `0x8db97c7cEcE249c2b98bDC0226Cc4C2A57BF52FC` is allowed to change the fee structure of the network. The admin address has to call the precompile in order to activate its effect; that is it needs to send a transaction with a new fee config to perform the update. This is a very powerful feature, but it also gives a large amount of power to the admin address. If the address `0x8db97c7cEcE249c2b98bDC0226Cc4C2A57BF52FC` is compromised, the network is compromised.

With the initial configurations, precompiles can immediately activate their effect on the activation timestamp. With this way admin addresses can be omitted from the precompile configuration. For example, the `feeManagerConfig` precompile can have `initialFeeConfig` to setup the fee configuration on the activation:

```

{
    "precompileUpgrades": [
        {
            "feeManagerConfig": {
                "blockTimestamp": 1668950000,
                "initialFeeConfig": {
                    "gasLimit": 20000000,
                    "targetBlockRate": 2,
                    "minBaseFee": 1000000000,
                    "targetGas": 100000000,
                    "baseFeeChangeDenominator": 48,
                    "minBlockGasCost": 0,
                    "maxBlockGasCost": 10000000,

```

```

        "blockGasCostStep": 500000
    }
}
]
}

```

Notice that there is no `adminAddresses` field in the configuration. This means that there will be no admin addresses to manage the fee structure with this precompile. The precompile will simply update the fee configuration to the specified fee config when it activates on the `blockTimestamp` `1668950000`.

:::note It's still possible to add `adminAddresses` or `enabledAddresses` along with these initial configurations. In this case, the precompile will be activated with the initial configuration, and admin/enabled addresses can access to the precompiled contract normally. :::

:::info

If you want to change the precompile initial configuration, you will need to first disable it then activate the precompile again with the new configuration.

:::

See every precompile initial configuration in their relevant `Initial Configuration` sections under [Precompiles](#).

AvalancheGo Chain Configs

As described in [this doc](#), each blockchain of Subnets can have its own custom configuration. If a Subnet's ChainID is `2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt`, the config file for this chain is located at `{chain-config-dir}/2ebCneCbwthjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt/config.json`.

For blockchains created by or forked from Subnet-EVM, most [C-Chain configs](#) are applicable except [Avalanche Specific APIs](#).

Priority Regossip

A transaction is "regossiped" when the node does not find the transaction in a block after `priority-regossip-frequency` (defaults to `1m`). By default, up to 16 transactions (max 1 per address) are regossiped to validators per minute.

Operators can use "priority regossip" to more aggressively "regossip" transactions for a set of important addresses (like bridge relayers). To do so, you'll need to update your [chain config](#) with the following:

```
{
  "priority-regossip-addresses": ["<YOUR 0x-ADDRESS>"]
}
```

By default, up to 32 transactions from priority addresses (max 16 per address) are regossipped to validators per second. You can override these defaults with the following config:

```
{
  "priority-regossip-frequency": "1s",
  "priority-regossip-max-txs": 32,
  "priority-regossip-addresses": ["<YOUR 0x-ADDRESS>"],
  "priority-regossip-txs-per-address": 16
}
```

Fee Recipient

This works together with [allowFeeRecipients](#) and [RewardManager precompile](#) to specify where the fees should be sent to.

With `allowFeeRecipients` enabled, validators can specify their addresses to collect fees.

```
{
  "feeRecipient": "<YOUR 0x-ADDRESS>"
}
```

:::warning

If `allowFeeRecipients` or [RewardManager precompile](#) is enabled on the Subnet, but a validator doesn't specify a "feeRecipient", the fees will be burned in blocks it produces.

:::

Network Upgrades: State Upgrades

Subnet-EVM allows the network operators to specify a modification to state that will take place at the beginning of the first block with a timestamp greater than or equal to the one specified in the configuration.

This provides a last resort path to updating non-upgradeable contracts via a network upgrade (for example, to fix issues when you are running your own blockchain).

:::warning

This should only be used as a last resort alternative to forking `subnet-evm` and specifying the network upgrade in code.

Using a network upgrade to modify state is not part of normal operations of the EVM. You should ensure the modifications do not invalidate any of the assumptions of deployed contracts or cause incompatibilities with downstream infrastructure such as block explorers.

•

The timestamps for upgrades in `stateUpgrades` must be in increasing order. `stateUpgrades` can be specified along with `precompileUpgrades` or by itself.

The following three state modifications are supported:

- `balanceChange` : adds a specified amount to the balance of a given account. This amount can be specified as hex or decimal and must be positive.
 - `storage` : modifies the specified storage slots to the specified values. Keys and values must be 32 bytes specified in hex, with a `0x` prefix.
 - `code` : modifies the code stored in the specified account. The code must *only* be the runtime portion of a code. The code must start with a `0x` prefix.

:::warning

If modifying the code, *only* the runtime portion of the bytecode should be provided in `upgrades.json`. Do not use the bytecode that would be used for deploying a new contract, as this includes the constructor code as well. Refer to your compiler's documentation for information on how to find the runtime portion of the contract you wish to modify.

• • •

The `upgrades.json` file shown below describes a network upgrade that will make the following state modifications at the first block after (or at) March 8, 2023 1:30:00 AM GMT :

- Sets the code for the account at `0x71562b71999873DB5b286dF957af199Ec94617F7`,
 - And adds `100` wei to the balance of the account at `0xb794f5ea0ba3949ce839613fffbba74279579268`,
 - Sets the storage slot `0x1234` to the value `0x6666` for the account at `0xb794f5ea0ba3949ce839613fffbba74279579268`

Deploy a Gnosis Safe on Your Subnet-EVM

Introduction

This article shows how to deploy and interact with a [Gnosis Safe](#) programmatically on any Subnet-EVM.

If you are looking for more information regarding the Gnosis Safe protocol, please check out [these developer docs](#).

Prerequisites

This tutorial assumes that:

- A Subnet and EVM blockchain has been created. Avalanche tools allow users to do this on [Mainnet](#), [Fuji](#) or a [Local network](#).
 - Your node is currently validating your target Subnet.
 - Your wallet has a balance of the Subnet native token (specified under `alloc` in your [Genesis File](#)).

The entirety of this tutorial will require you to work with 3 projects (4 if running locally)

- [gnosis-Subnet](#)
 - [safe-tasks](#)
 - [avalanche-smart-contract-quickstart](#)
 - [avalanche-network-runner](#) (Local Workflow)

Custom Network Workflow

Setup Network

In order for the `gnosis-safe` repo to successfully deploy these contracts, ensure that you have `jq` and `yarn` installed.

On Ubuntu, run `sudo apt install jq`, `sudo apt install yarn`. On Linux, run `brew install jq`, `brew install yarn`.

Next, clone the library. Change `.env.example` to `.env` and set the variable, `MNEMONIC` to the seed phrase of the wallet you intend to deploy the contracts with. Set `ADDRESS` to the public key of the same wallet. Finally, set `NODE_URL` to the URL of your Subnet's RPC.

:::note

This address you choose must be funded as the transactions will include a gas fee.

:::

Example:

```
export MNEMONIC="foo bar foo bar"
export ADDRESS="0xA028036b2aaAED2487654B2B042C2AA9FA5Ef6b8"
export NODE_URL="https://api.avax-test.network/ext/bc/C/rpc"
```

Deploy the Safe Contracts

After setting up the `.env` file and installing `jq`, simply run `./deploy.sh`.

```
deploying "SimulateTxAccessor" (tx: 0xb2104e7067e35e1d2176ee53f6030bbcef4a12051505daca603d097d87ebd3e2) ...: deployed at 0x52C84043CD9c865236f11d9Fc9F56aa003c1f922 with 237301 gas
deploying "GnosisSafeProxyFactory" (tx: 0x8faec24dda341141e02d1b898ceeffe445b2893b3f600f1f79a5e04e3a91396cd) ...: deployed at 0x17aB05351Fc94a1a67Bf3f56Ddb8941aE6c63E25 with 865918 gas
deploying "DefaultCallbackHandler" (tx: 0xala48e8869c71cb10e9ca5f2ce20420c44ce09dc32aa13efbd2ebc3796bcf145) ...: deployed at 0x5aa01B3b5877255cE50cc55e8986a7a5fe29C70e with 541390 gas
deploying "CompatibilityFallbackHandler" (tx: 0x05d1f9ef7cafdf2dbc5d4b9621d15e15f2416e6917371355718e6194d6e39871a) ...: deployed at 0x5DB9A7629912EBF95876228C24A848de0bfB43A9 with 1235752 gas
deploying "CreateCall" (tx: 0xb240c1594dc5cd1a37b8890e2a0e610c0339af157d094d008e8eebcf3eb3275) ...: deployed at 0x4Ac1d98D9cEF99EC6546dEd48d550b0b287aaD6D with 294075 gas
deploying "MultiSend" (tx: 0x075067ca5e4755c31e8dbe5e16cd597f86fb141f45de254d39b050568ef2a3a6) ...: deployed at 0xA4cD3b0Eb6E5Ab5d8CE4065BccD70040ADAB1F00 with 189518 gas
deploying "MultiSendCallOnly" (tx: 0xa237e18fb2561c2081341f3621ff559063bd07c6b9f77aefdaf103f976751353) ...: deployed at 0xa4DfF80B4a1D748BF28BC4A271eB834689Ea3407 with 141745 gas
deploying "SignMessageLib" (tx: 0x1cc1322268015fee470529682dbc9fc8aa068554df841de824524cdfb8bc2e8) ...: deployed at 0xe336d36facA76840407e6836d2619E1Ec0A2b4 with 261758 gas
deploying "GnosisSafeL2" (tx: 0x341ec664d3a5c2c98f1c3f5862651ba82e0c2d12875d69ad3bdf8d1d5e3e074b) ...: deployed at 0x95CA0a568236fc7413Cd2b794A7da24422c2BBb6 with 5268965 gas
deploying "GnosisSafe" (tx: 0x10dcfc8c5f53ae698c77d7f60d6756b4b24f2f8224e14e21658c421e158a84cd4) ...: deployed at 0x789a5FDac2b37FCD290fb2924382297A6AE65860 with 5086960 gas
👉 Done in 26.90s.
```

Not all contracts will deploy, but that is expected behavior. If you see this output, everything worked as expected:

```
Verification status for CompatibilityFallbackHandler: SUCCESS
Verification status for CreateCall: SUCCESS
Verification status for DefaultCallbackHandler: SUCCESS
Verification status for GnosisSafe: SUCCESS
Verification status for GnosisSafeL2: SUCCESS
Verification status for GnosisSafeProxyFactory: SUCCESS
Verification status for MultiSend: FAILURE
Verification status for MultiSendCallOnly: SUCCESS
Verification status for SignMessageLib: SUCCESS
Verification status for SimulateTxAccessor: FAILURE
```

Deployment information, including contract addresses can be found in `safe-contracts/deployments/custom`.

:::note Please record your GnosisSafeL2 and GnosisSafeProxyFactory to be able to create a [Safe](#) on your Subnet. :::

The deployment of the contracts is using a [proxy factory](#), therefore the address is depending on the bytecode. If the address is the same then the deployment bytecode of the contract is also the same (assuming that the target chain follows the EVM specifications set in the Ethereum Yellow Paper).

Deploy a Smart Contract on Your Subnet-EVM Using Remix and MetaMask

Introduction

This tutorial assumes that:

- [An Subnet and EVM blockchain](#) has been created
- Your Node is currently validating your target Subnet
- Your wallet has a balance of the Subnet Native Token(Specified under `alloc` in your [Genesis File](#)).

Step 1: Setting up MetaMask

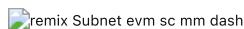
EVM Subnet Settings: ([EVM MetaMask Tutorial](#)).

- `Network Name` : Custom Subnet-EVM

- **New RPC URL** : <http://NodeIPAddress:9650/ext/bc/BlockchainID/rpc> (Note: the port number should match your local setting which can be different from 9650.)
- **ChainID** : Subnet-EVM ChainID
- **Symbol** : Subnet-EVM Token Symbol
- **Explorer** : N/A



You should see a balance of your Subnet's Native Token in MetaMask.



Step 2: Connect MetaMask and Deploy a Smart Contract

Using Remix

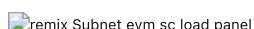
Open [Remix](#) -> Select Solidity.



Create the smart contracts that we want to compile and deploy using Remix file explorer

Using GitHub

In Remix Home Click the GitHub button.



Paste the [link to the Smart Contract](#) into the popup and Click import.



For this example, we will deploy an ERC721 contract from the [Avalanche Smart Contract Quickstart Repository](#).



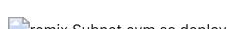
Navigate to Deploy Tab -> Open the "ENVIRONMENT" drop-down and select Injected Web3 (make sure MetaMask is loaded).



Once we injected the web3-> Go back to the compiler, and compile the selected contract -> Navigate to Deploy Tab.



Now, the smart contract is compiled, MetaMask is injected, and we are ready to deploy our ERC721. Click "Deploy."



Confirm the transaction on the MetaMask pop up.



Our contract is successfully deployed!



Now, we can expand it by selecting it from the "Deployed Contracts" tab and test it out.



The contract ABI and Bytecode are available on the compiler tab.



If you had any difficulties following this tutorial or simply want to discuss Avalanche with us, you can join our community at [Discord](#)!

Other Tools

You can use Subnet-EVM just like you use C-Chain and EVM tools. Only differences are `chainID` and RPC URL. For example you can deploy your contracts with [hardhat quick start guide](#) by changing `url` and `chainId` in the `hardhat.config.ts`.

Deploying Cross-Chain EVM <-> EVM Bridge

:::warning

This tutorial is for demo purpose on how to build a cross-chain bridge. It is not for production use. You must take the full responsibility to ensure your bridge's security.

:::

Introduction

In this tutorial, we will be building a bridge between [WAGMI](#) and [Fuji](#). This bridge will help us to transfer native **WGM** coin wrapped into **WWGM** back and forth from the WAGMI chain to the Fuji chain. Using this guide, you can deploy a bridge between any EVM-based chains for any ERC20 tokens.

The wrapped version of a native coin is its pegged ERC20 representation. Wrapping it with the ERC20 standard makes certain processes like delegated transactions much easier. You can easily get wrapped tokens by sending the native coin to the wrapped token contract address.

WAGMI is an independent EVM-based test chain deployed on a custom Subnet on the Avalanche network.

We will be using [ChainSafe](#)'s bridge repository, to easily set up a robust and secure bridge.

Workflow of the Bridge

WAGMI and Fuji chains are not interconnected by default, however, we could make them communicate. Relayers watch for events (by polling blocks) on one chain and perform necessary action using those events on the other chain. This way we can also perform bridging of tokens from one chain to the other chain through the use of smart contracts.

Here is the basic high-level workflow of the bridge -

- Users deposit token on the Bridge contract
- Bridge contract asks Handler contract to perform deposit action
- Handler contract **locks** the deposited token in the token safe
- Bridge contract emits `Deposit` event
- Relayer receives the `Deposit` event from the source chain
- Relayer creates a voting proposal on the destination chain to mint a new token
- After threshold relayer votes, the proposal is executed
- Tokens are **minted** to the recipient's address

Bridging tokens from source to destination chain involves the **lock and mint** approach. Whereas bridging tokens from destination to source chain involves **burn and release** approach. We cannot mint and burn tokens that we do not control. Therefore we lock them in the token safe on the source chain. And mint the corresponding token (which we will deploy and hence control) on the destination chain.



Requirements

These are the requirement to follow this tutorial -

- Add [WAGMI](#) and [Fuji](#) chain on the MetaMask network
- Import `WWGM` token (asset) on the WAGMI network (MetaMask). Here is the address - `0x3Ee7094DADda15810F191DD6AcP7E4FFa37571e4`
- `WGM` coins on the WAGMI chain. Drip `1 WGM` from the [WAGMI Faucet](#).
- `AVAX` coins on the Fuji chain. Drip `10 AVAX` from the [Fuji Faucet](#).
- Wrapped `WGM` tokens on the WAGMI chain. Send a few `WGM` coins to the `WWGM` token address (see second point), to receive the same amount of `WWGM`. Always keep some `WGM` coins, to cover transaction fees.

Setting Up Environment

Let's make a new directory `deploy-bridge`, where we will be keeping our bridge codes. We will be using the following repositories -

- [ChainSafe/chainbridge-deploy](#) - This will help us in setting up of our bridge contracts
- [ChainSafe/ChainBridge](#) - This will help us in setting up of our off-chain relayer.

Installing ChainBridge Command-Line Tool

Using the following command, we can clone and install ChainBridge's command-line tool. This will help us in setting up bridge contracts and demonstrating bridge transfers. Once the bridge contracts are deployed, you can use its ABI and contract address to set up your UI.

```
git clone -b v1.0.0 --depth 1 https://github.com/ChainSafe/chainbridge-deploy \
&& cd chainbridge-deploy/cb-sol-cli \
&& npm install \
&& make install
```

This will build the contracts and installs the `cb-sol-cli` command.

Setting Up Environment Variables

Let's set up environment variables, so that, we do not need to write their values every time we issue a command. Move back to the `deploy-bridge` directory (main project directory) and make a new file `configVars`. Put the following contents inside it -

```
SRC_GATEWAY=https://subnets.avax.network/wagmi-wagmi-chain-testnet/rpc
DST_GATEWAY=https://api.avax-test.network/ext/bc/C/rpc

SRC_ADDR=<Your address on WAGMI>
SRC_PK=<your private key on WAGMI>
DST_ADDR=<Your address on Fuji>
DST_PK=<your private key on Fuji>

SRC_TOKEN="0x3Ee7094DAdda15810F191DD6AcF7E4FFa37571e4"
RESOURCE_ID="0x00"
```

- `SRC_ADDR` and `DST_ADDR` are the addresses that will deploy bridge contracts and will act as a relayer.
- `SRC_TOKEN` is the token that we want to bridge. Here is the address of the wrapped ERC20 version of the WGM coin aka wWGM.
- `RESOURCE_ID` could be anything. It identifies our bridged ERC20 tokens on both sides (WAGMI and Fuji).

Every time we make changes to these config variables, we have to update our bash environment. Run the following command according to the relative location of the file. These variables are temporary and are only there in the current terminal session, and will be flushed, once the session is over. Make sure to load these environment variables anywhere you will using them in the bash commands (like `$SRC_GATEWAY` or `$SRC_ADDR`)

```
source ./configVars
```

Setting Up Source Chain

We need to set up our source chain as follows -

- Deploy Bridge and Handler contract with `$SRC_ADDR` as default and only relayer
- Register the `wWGM` token as a resource on the bridge

Deploy Source Contracts

The command-line tool `cb-sol-cli` will help us to deploy the contracts. Run the following command in the terminal session where the config vars are loaded. It will add `SRC_ADDR` as the default relayer for relaying events from the WAGMI chain (source) to the Fuji chain (destination).

One of the most important parameter to take care of while deploying bridge contract is the `expiry` value. It is the number of blocks after which a proposal is considered cancelled. By default it is set to 100. On Avalanche Mainnet, with this value, the proposals could be expired within 3-4 minutes. You should choose a very large expiry value, according to the chain you are deploying bridge to. Otherwise your proposal will be cancelled if the threshold number of vote proposals are not received on time.

You should also keep this in mind that sometimes during high network activity, a transaction could stuck for a long time. Proposal transactions stuck in this scenario, could result in the cancellation of previous proposals. Therefore, expiry values should be large enough, and relayers should issue transactions with a competitive max gas price.

```
cb-sol-cli --url $SRC_GATEWAY --privateKey $SRC_PK --gasPrice 25000000000 deploy \
--bridge --erc20Handler \
--relayers $SRC_ADDR \
--relayerThreshold 1 \
--expiry 500 \
--chainId 0
```

The output will return deployed contracts' (Bridge and Handler) address. Update the `configVars` file with these addresses by adding the following 2 variables and loading them to the environment.

```
SRC_BRIDGE=<resulting bridge contract address>
SRC_HANDLER=<resulting erc20 handler contract address>
```

Make sure to load these using the `source` command.

Configure Resource on Bridge

Run the following command to register the `wWGM` token as a resource on the source bridge.

```
cb-sol-cli --url $SRC_GATEWAY --privateKey $SRC_PK --gasPrice 25000000000 bridge register-resource \
--bridge $SRC_BRIDGE \
--handler $SRC_HANDLER \
--resourceId $RESOURCE_ID \
--targetContract $SRC_TOKEN
```

Setting Up Destination Chain

We need to set up our destination chain as follows -

- Deploy Bridge and Handler contract with `$DST_ADDR` as default and only relayer

- Deploy mintable and burnable ERC20 contract representing bridged `wWGM` token
- Register the `wWGM` token as a resource on the bridge
- Register the `wWGM` token as mintable/burnable on the bridge
- Giving permissions to Handler contract to mint new `wWGM` tokens

Deploy Destination Contracts

Run the following command to deploy Bridge, ERC20 Handler, and `wWGM` token contracts on the Fuji chain. Again it will set `DST_ADDR` as the default relayer for relaying events from Fuji chain (destination) to WAGMI chain (source). For this example, both `SRC_ADDR` and `DST_ADDR` represent the same thing.

```
cb-sol-cli --url $DST_GATEWAY --privateKey $DST_PK --gasPrice 25000000000 deploy\
--bridge --erc20 --erc20Handler \
--relayers $DST_ADDR \
--relayerThreshold 1 \
--chainId 1
```

Update the environment variables with the details which you will get by running the above command. Don't forget to load these variables.

```
DST_BRIDGE=<resulting bridge contract address>
DST_HANDLER=<resulting erc20 handler contract address>
DST_TOKEN=<resulting erc20 token address>
```

Configuring Resource on Bridge

Run the following command to register deployed `wWGM` token as a resource on the bridge.

```
cb-sol-cli --url $DST_GATEWAY --privateKey $DST_PK --gasPrice 25000000000 bridge register-resource \
--bridge $DST_BRIDGE \
--handler $DST_HANDLER \
--resourceId $RESOURCE_ID \
--targetContract $DST_TOKEN
```

Setting Token as Mintable and Burnable on Bridge

The bridge has two options when it receives a deposit of a token -

- Lock the received token on one chain and mint the corresponding token on the other chain
- Burn the received token on one chain and release the corresponding token on the other chain

We cannot mint or burn any token which we do not control. Though we can lock and release such tokens by putting them in a token safe. The bridge has to know which token it can burn. With the following command, we can set the resource as burnable. The bridge will choose the action accordingly, by seeing the token as burnable or not.

```
cb-sol-cli --url $DST_GATEWAY --privateKey $DST_PK --gasPrice 25000000000 bridge set-burn \
--bridge $DST_BRIDGE \
--handler $DST_HANDLER \
--tokenContract $DST_TOKEN
```

Authorizing Handler to Mint New Tokens

Now let's permit the handler to mint the deployed ERC20 (`wWGM`) token on the destination chain. Run the following command.

```
cb-sol-cli --url $DST_GATEWAY --privateKey $DST_PK --gasPrice 25000000000 erc20 add-minter \
--minter $DST_HANDLER \
--erc20Address $DST_TOKEN
```

The deployer of the contracts (here `SRC_ADDR` or `DST_ADDR`) holds the admin rights. An admin can add or remove a new relayer, minter, admin etc. It can also mint new ERC20 tokens on the destination chain. You can issue these commands using `cb-sol-cli` with the options mentioned in these files. The mint command should not be used manually, unless some intervention is required, when the relayers failed to mint the tokens on the destination chain on time.

Deploy Relayer

All the on-chain setups like deploying bridges, handlers, tokens, etc. are complete. But the two chains are not interconnected. We need some off-chain relayer to communicate messages between the chains. The relayer will poll for deposit events on one chain, and submit vote proposals to mint or release the corresponding token on another chain.

Since we set the relayer threshold to 1, while deploying the bridge and handler, we require a voting proposal from only 1 relayer. But in production, we should use a large set of relayers with a high threshold to avoid power concentration.

For this purpose, we will be using ChainSafe's relayer. Follow the steps described below to deploy the relayer.

Cloning and Building Relayer

Open a new terminal session, while keeping the previous session loaded with environment variables. We have to load the environment variables in this session too. Load these variables in this session too using the `source` command.

Now, move to the `deploy-bridge` directory and run the following command to clone the relayer repository (implemented in Go), and build its binary.

```
git clone -b v1.1.1 --depth 1 https://github.com/ChainSafe/chainbridge \
&& cd chainbridge \
&& make build
```

This will create a binary inside the `chainbridge/build` directory as `chainbridge`.

Configuring Relayer

The relayer requires some configurations like source chain, destination chain, bridge, handler address, etc. Run the following command. It will make a `config.json` file with the required details in it. You can update these details, as per your need.

```
echo "{
  \"chains\": [
    {
      \"name\": \"WAGMI\",
      \"type\": \"ethereum\",
      \"id\": \"0\",
      \"endpoint\": \"$SRC_GATEWAY\",
      \"from\": \"$SRC_ADDR\",
      \"opts\": {
        \"bridge\": \"$SRC_BRIDGE\",
        \"erc20Handler\": \"$SRC_HANDLER\",
        \"genericHandler\": \"$SRC_HANDLER\",
        \"gasLimit\": \"1000000\",
        \"maxGasPrice\": \"5000000000\",
        \"http\": \"true\",
        \"blockConfirmations\": \"0\"
      }
    },
    {
      \"name\": \"Fuji\",
      \"type\": \"ethereum\",
      \"id\": \"1\",
      \"endpoint\": \"$DST_GATEWAY\",
      \"from\": \"$DST_ADDR\",
      \"opts\": {
        \"bridge\": \"$DST_BRIDGE\",
        \"erc20Handler\": \"$DST_HANDLER\",
        \"genericHandler\": \"$DST_HANDLER\",
        \"gasLimit\": \"1000000\",
        \"maxGasPrice\": \"5000000000\",
        \"http\": \"true\",
        \"blockConfirmations\": \"0\"
      }
    }
  ]
}" >> config.json
```

Check and confirm the details in the `config.json` file.

In the above command, you can see that `blockConfirmations` is set to `0`. This will work well for networks like Avalanche because the block is confirmed once it's committed. Unlike other chains such as Ethereum, which requires 20-30 block confirmations. Therefore, use this configuration with caution, depending on the type of chain you are using.

It can cause serious problems if a corresponding token is minted or released based on an unconfirmed block.

Set Up Keys

Give relayer access to your keys. Using these keys, the relayer will propose deposit events and execute proposals. It will ask to set a password for encrypting these keys. Every time you start the relayer, it will ask for this password.

```
./build/chainbridge accounts import --privateKey $SRC_PK
```

```
./build/chainbridge accounts import --privateKey $DST_PK
```

Let's Test the Bridge

The setup is now complete - both on-chain and off-chain. Now we just have to start the relayer and test the bridge. For testing purposes, we will be using `cb-sol-cli` to make deposit transactions on the bridge. But you can make your frontend and integrate it with the bridge using the ABIs.

Start Relayer

Run the following command to start the relayer. It will print logs of all the events associated with our bridge, happening on both the chains. So keep the relayer running and follow the next commands in the other terminal session.

```
./build/chainbridge --config config.json --verbosity trace --latest
```

Approve Handler to Spend my Tokens

Now, let's deposit tokens on the WAGMI bridge. But before that, we need to approve the handler to spend (lock or burn) tokens on our (here `SRC_PK`) behalf. The amount here is in Wei (1 ether (WGM) = 10^{18} Wei). We will be approving 0.1 wWGM.

```
cb-sol-cli --url $SRC_GATEWAY --privateKey $SRC_PK --gasPrice 25000000000 erc20 approve \
    --amount 1000000000000000000 \
    --erc20Address $SRC_TOKEN \
    --recipient $SRC_HANDLER
```

Deposit Tokens to the Bridge

Once approved, we can send a deposit transaction. Now let's deposit 0.1 wWGM on the bridge. The handler will lock (transfer to token safe) 0.1 wWGM from our address (here `SRC PK`) and mint the new tokens on the destination chain to the recipient (here `DST ADDR`).

```
cb-sol-cli --url $SRC_GATEWAY --privateKey $SRC_PK --gasPrice 25000000000 erc20 deposit \
--amount 100000000000000000 \
--dest 1 \
--bridge $SRC_BRIDGE \
--recipient $DST_ADDR \
--resourceid $RESOURCE_ID
```

This transaction will transfer 0.1 wWGM to token safe and emit a `Deposit` event, which will be captured by the relayer. Following this event, it will send a voting proposal to the destination chain. Since the threshold is 1, the bridge will execute the proposal, and new wWGM minted to the recipient's address. Here is the screenshot of the output from the relayer.



Similarly, we can transfer the tokens back to the WAGMI chain.

Conclusion

Similar to the above process, you can deploy a bridge between any 2 EVM-based chains. We have used the command-line tool to make approvals and deposits. This can be further extended to build frontend integrated with the bridge. Currently, it depends on a single relayer, which is not secure. We need a large set of relayers and a high threshold to avoid any kind of centralization.

You can learn more about these contracts and implementations by reading ChainSafe's [ChainBridge](#) documentation.

Deploy Subnets on Production Infrastructure

Introduction

After architecting your Subnet environment on the [local machine](#), proving the design and testing it out on the [Fuji Testnet](#), eventually you will need to deploy your Subnet to production environment. Running a Subnet in production is much more involved than local and Testnet deploys, as your Subnet will have to take care of real world usage, maintaining uptime, upgrades and all of that in a potentially adversarial environment. The purpose of this document is to point out a set of general considerations and propose potential solutions to them.

The architecture of the environment your particular Subnet will use will be greatly influenced by the type of load and activity your Subnet is designed to support so your solution will most likely differ from what we propose here. Still, it might be useful to follow along, to build up the intuition for the type of questions you will need to consider.

Node Setup

Avalanche nodes are essential elements for running your Subnet in production. At a minimum, your Subnet will need validator nodes, potentially also nodes that act as RPC servers, indexers or explorers. Running a node is basically running an instance of [AvalancheGo](#) on a server.

Server OS

Although AvalancheGo can run on a macOS or a Windows computer, we strongly recommend running nodes on computers running Linux as they are designed specifically for server loads and all the tools and utilities needed for administering a server are native to Linux.

Hardware Specification

For running AvalancheGo as a validator on the Primary Network the recommended configuration is as follows:

- CPU: Equivalent of 8 AWS vCPU
 - RAM: 16 GiB
 - Storage: 1 TiB with at least 3000 IOPS
 - OS: Ubuntu 20.04
 - Network: Reliable IPv4 or IPv6 network connection, with an open public port

That is the configuration sufficient for running a Primary Network node. Any resource requirements for your Subnet come on top of this, so you should not go below this configuration, but may need to step up the specification if you expect your Subnet to handle a significant amount of transactions.

Be sure to set up monitoring of resource consumption for your nodes because resource exhaustion may cause your node to slow down or even halt, which may severely impact your Subnet negatively.

Server Location

You can run a node on a physical computer that you own and run, or on a cloud instance. Although running on your own HW may seem like a good idea, unless you have a sizeable DevOps 24/7 staff we recommend using cloud service providers as they generally provide reliable computing resources that you can count on to be properly maintained and monitored.

Local Servers

If you plan on running nodes on your own hardware, make sure they satisfy the minimum HW specification as outlined earlier. Pay close attention to proper networking setup, making sure the p2p port (9651) is accessible and public IP properly configured on the node. Make sure the node is connected to the network physically (not over Wi-Fi), and that the router is powerful enough to handle a couple of thousands of persistent TCP connections and that network bandwidth can accommodate at least 5Mbps of steady upstream and downstream network traffic.

When installing the AvalancheGo node on the machines, unless you have a dedicated DevOps staff that will take care of node setup and configuration, we recommend using the [installer script](#) to set up the nodes. It will abstract most of the setup process for you, set up the node as a system service and will enable easy node upgrades.

Cloud Providers

There are a number of different cloud providers. We have documents that show how to set up a node on the most popular ones:

- [Amazon Web Services](#)
- [Azure](#)
- [Google Cloud Platform](#)

There is a whole range of other cloud providers that may offer lower prices or better deals for your particular needs, so it makes sense to shop around.

Once you decide on a provider (or providers), if they offer instances in multiple data centers, it makes sense to spread the nodes geographically since that provides a better resilience and stability against outages.

Number of Validators

Number of validators on a Subnet is a crucial decision you need to make. For stability and decentralization, you should strive to have as many validators as possible.

For stability reasons our recommendation is to have **at least 5** full validators on your Subnet. If you have less than 5 validators your Subnet liveness will be at risk whenever a single validator goes offline, and if you have less than 4 even one offline node will halt your Subnet.

You should be aware that 5 is the minimum we recommend. But, from a decentralization standpoint having more validators is always better as it increases the stability of your Subnet and makes it more resilient to both technical failures and adversarial action. In a nutshell: run as many Subnet validators as you can.

Considering that at times you will have to take nodes offline, for routine maintenance (at least for node upgrades which happen with some regularity) or unscheduled outages and failures you need to be able to routinely handle at least one node being offline without your Subnet performance degrading.

Node Bootstrap

Once you set up the server and install AvalancheGo on them, nodes will need to bootstrap (sync with the network). This is a lengthy process, as the nodes need to catch up and replay all the network activity since the genesis up to the present moment. Full bootstrap on a node can take more than a week, but there are ways to shorten that process, depending on your circumstances.

State Sync

If the nodes you will be running as validators don't need to have the full transaction history, then you can use [state sync](#). With this flag enabled, instead of replaying the whole history to get to the current state, nodes simply download only the current state from other network peers, shortening the bootstrap process from multiple days to a couple of hours. If the nodes will be used for Subnet validation exclusively, you can use the state sync without any issues. Currently, state sync is only available for the C-Chain, but since the bulk of the transactions on the platform happen there it still has a significant impact on the speed of bootstrapping.

Database Copy

Good way to cut down on bootstrap times on multiple nodes is database copy. Database is identical across nodes, and as such can safely be copied from one node to another. Just make sure to that the node is not running during the copy process, as that can result in a corrupted database. Database copy procedure is explained in detail [here](#).

Please make sure you don't reuse any node's NodeID by accident, especially don't restore another node's ID, see [here](#) for details. Each node must have its own unique NodeID, otherwise, the nodes sharing the same ID will not behave correctly, which will impact your validator's uptime, thus staking rewards, and the stability of your Subnet.

Subnet Deploy

Once you have the nodes set up you are ready to deploy the actual Subnet. Right now, the recommended tool to do that is [Avalanche-CLI](#).

Instructions for deployment by Avalanche-CLI can be found [here](#).

Ledger HW Wallet

When creating the Subnet, you will be required to have a private key that will control the administrative functions of the Subnet (adding validators, managing the configuration). Needless to say, whoever has this private key has complete control over the Subnet and the way it runs. Therefore, protecting that key is of the utmost operational importance. Which is why we strongly recommend using a hardware wallet such as a [Ledger HW Wallet](#) to store and access that private key.

General instruction on how to use a Ledger device with Avalanche can be found [here](#).

Genesis File

The structure that defines the most important parameters in a Subnet is found in the genesis file, which is a `json` formatted, human-readable file. Describing the contents and the options available in the genesis file is beyond the scope of this document, and if you're ready to deploy your Subnet to production you probably have it mapped out already.

If you want to review, we have a description of the genesis file in our document on [customizing EVM Subnets](#).

Validator Configuration

Running nodes as Subnet validators warrants some additional considerations, above those when running a regular node or a Primary Network-only validator.

Joining a Subnet

For a node to join a Subnet, there are two prerequisites:

- Primary Network validation
- Subnet tracking

Primary Network validation means that a node cannot join a Subnet as a validator before becoming a validator on the Primary Network itself. So, after you add the node to the validator set on the Primary Network, node can join a Subnet. Of course, this is valid only for Subnet validators, if you need a non-validating Subnet node, then the node doesn't need to be a validator at all.

To have a node start syncing the Subnet, you need to add the `--track-subnets` command line option, or `track-subnets` key to the node config file (found at `~/.avalanchego/configs/node.json` for installer-script created nodes). A single node can sync multiple Subnets, so you can add them as a comma-separated list of Subnet IDs.

An example of a node config syncing two Subnets:

```
{  
  "public-ip-resolution-service": "opendns",  
  "http-host": "",  
  "track-subnets": "28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY,Ai42MkKqk8yjXFCpoHXw7rdTWSHiKEMqh5h8gbxwjgkCUfkkrk"  
}
```

But that is not all. Besides tracking the SubnetID, the node also needs to have the plugin that contains the VM instance the blockchain in the Subnet will run. You should have already been through that on Testnet and Fuji, but for a refresher, you can refer to [this tutorial](#).

So, name the VM plugin binary as the `VMID` of the Subnet chain and place it in the `plugins` directory where the node binary is (for installer-script created nodes that would be `~/avalanche-node/plugins/`).

Subnet Bootstrapping

After you have tracked the Subnet and placed the VM binary in the correct directory, your node is ready to start syncing with the Subnet. Restart the node and monitor the log output. You should notice something similar to:

```
Jul 30 18:26:31 node-fuji avalanchego[1728308]: [07-30|18:26:31.422] INFO chains/manager.go:262 creating chain:  
Jul 30 18:26:31 node-fuji avalanchego[1728308]: ID: 2ebCneCbwhjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt  
Jul 30 18:26:31 node-fuji avalanchego[1728308]: VMID:srEXiWaHuhNyGwPUI444Tu47ZEDwxTWRbQiuD7FmgSAQ6X7Dy
```

That means the node has detected the Subnet, and is attempting to initialize it and start bootstrapping the Subnet. It might take some time (if there are already transactions on the Subnet), and eventually it will finish the bootstrap with a message like:

```
Jul 30 18:27:21 node-fuji avalanchego[1728308]: [07-30|18:27:21.055] INFO <2ebCneCbwhjQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8geqh6tt  
Chain> snowman/transitive.go:333 consensus starting with J5wjnotMCrM2DKxeBTBpfwgCPpvstugWNozLog2TomTjSuGK as the last accepted  
block
```

That means the node has successfully bootstrapped the Subnet and is now in sync. If the node is one of the validators, it will start validating any transactions that get posted to the Subnet.

Monitoring

If you want to inspect the process of Subnet syncing, you can use the RPC call to check for the [blockchain status](#).

For a more in-depth look into Subnet operation, check out the blockchain log. By default, the log can be found in `~/.avalanchego/logs/ChainID.log` where you replace the `ChainID` with the actual ID of the blockchain in your Subnet.

For an even more thorough (and pretty!) insight into how the node and the Subnet is behaving, you can install the Prometheus+Grafana monitoring system with the custom dashboards for the regular node operation, as well as a dedicated dashboard for Subnet data. Check out the [tutorial](#) for information on how to set it up.

Managing Validation

On Avalanche all validations are limited in time and can range from two weeks up to one year. Furthermore, Subnet validations are always a subset of the Primary Network validation period (must be shorter or the same). That means that periodically your validators will expire and you will need to submit a new validation transaction for both the Primary Network and your Subnet.

Unless managed properly and in a timely manner, that can be disruptive for your Subnet (if all validators expire at the same time your Subnet will halt). To avoid that, keep notes on when a particular validation is set to expire and be ready to renew it as soon as possible. Also, when initially setting up the nodes, make sure

to stagger the validator expiry so they don't all expire on the same date. Setting end dates at least a day apart is a good practice, as well as setting reminders for each expiry.

Conclusion

Hopefully, by reading this document you have a better picture of the requirements and considerations you need to make when deploying your Subnet to production and you are now better prepared to launch your Subnet successfully.

Keep in mind, running a Subnet in production is not a one-and-done kind of situation, it is in fact running a fleet of servers 24/7. And as with any real time service, you should have a robust logging, monitoring and alerting systems to constantly check the nodes and Subnet health and alert you if anything out of the ordinary happens.

If you have any questions, doubts or would like to chat, please check out our [Discord server](#), where we host a dedicated `#subnet-chat` channel dedicated to talking about all things Subnet.

We hope to see you there!

Stateful Precompile Generation Tutorial

In this tutorial, we are going to walk through how we can generate a stateful precompile from scratch. Before we start, let's brush up on what a precompile is, what a stateful precompile is, and why this is extremely useful.

Background

Precompiled Contracts

Ethereum uses precompiles to efficiently implement cryptographic primitives within the EVM instead of re-implementing the same primitives in Solidity. The following precompiles are currently included: `ecrecover`, `sha256`, `blake2f`, `ripemd160`, `Bn256Add`, `Bn256Mul`, `Bn256Pairing`, the identity function, and modular exponentiation.

We can see these [precompile](#) mappings from address to function here in the Ethereum VM:

```
// PrecompiledContractsBerlin contains the default set of pre-compiled Ethereum
// contracts used in the Berlin release.
var PrecompiledContractsBerlin = map[common.Address]PrecompiledContract{
    common.BytesToAddress([]byte{1}): &ecrecover{},
    common.BytesToAddress([]byte{2}): &sha256hash{},
    common.BytesToAddress([]byte{3}): &ripemd160hash{},
    common.BytesToAddress([]byte{4}): &dataCopy{},
    common.BytesToAddress([]byte{5}): &bigModExp{eip256s: true},
    common.BytesToAddress([]byte{6}): &bn256AddIstanbul{},
    common.BytesToAddress([]byte{7}): &bn256ScalarMulIstanbul{},
    common.BytesToAddress([]byte{8}): &bn256PairingIstanbul{},
    common.BytesToAddress([]byte{9}): &blake2F{},
}
```

These precompile addresses start from `0x0001` and increment by 1.

A [precompile](#) follows this interface:

```
// PrecompiledContract is the basic interface for native Go contracts. The implementation
// requires a deterministic gas count based on the input size of the Run method of the
// contract.
type PrecompiledContract interface {
    RequiredGas(input []byte) uint64 // RequiredPrice calculates the contract gas use
    Run(input []byte) ([]byte, error) // Run runs the precompiled contract
}
```

Here is an example of the [sha256 precompile](#) function.

```
type sha256hash struct{}

// RequiredGas returns the gas required to execute the pre-compiled contract.
//
// This method does not require any overflow checking as the input size gas costs
// required for anything significant is so high it's impossible to pay for.
func (c *sha256hash) RequiredGas(input []byte) uint64 {
    return uint64(len(input)+31)/32*params.Sha256PerWordGas + params.Sha256BaseGas
}

func (c *sha256hash) Run(input []byte) ([]byte, error) {
    h := sha256.Sum256(input)
    return h[:], nil
}
```

The CALL opcode (CALL, STATICCALL, DELEGATECALL, and CALLCODE) allows us to invoke this precompile.

The function signature of CALL in the EVM is as follows:

```
Call(  
    caller ContractRef,  
    addr common.Address,  
    input []byte,  
    gas uint64,  
    value *big.Int,  
) (ret []byte, leftOverGas uint64, err error)
```

Precompiles are a shortcut to execute a function implemented by the EVM itself, rather than an actual contract. A precompile is associated with a fixed address defined in the EVM. There is no byte code associated with that address.

When a precompile is called, the EVM checks if the input address is a precompile address, and if so it executes the precompile. Otherwise, it loads the smart contract at the input address and runs it on the EVM interpreter with the specified input data.

Stateful Precompiled Contracts

A stateful precompile builds on a precompile in that it adds state access. Stateful precompiles are not available in the default EVM, and are specific to Avalanche EVMS such as [Coreth](#) and [Subnet-EVM](#).

A stateful precompile follows this [interface](#):

```
// StatefulPrecompiledContract is the interface for executing a precompiled contract  
type StatefulPrecompiledContract interface {  
    // Run executes the precompiled contract.  
    Run(accessibleState PrecompileAccessibleState,  
        caller common.Address,  
        addr common.Address,  
        input []byte,  
        suppliedGas uint64,  
        readOnly bool)  
    (ret []byte, remainingGas uint64, err error)  
}
```

A stateful precompile injects state access through the `PrecompileAccessibleState` interface to provide access to the EVM state including the ability to modify balances and read/write storage.

This way we can provide even more customization of the EVM through Stateful Precompiles than we can with the original precompile interface!

AllowList

The AllowList enables a precompile to enforce permissions on addresses. The AllowList is not a contract itself, but a helper structure to provide a control mechanism for wrapping contracts. It provides an `AllowListConfig` to the precompile so that it can take an initial configuration from genesis/upgrade. It also provides 4 functions to set/read the permissions. In this tutorial, we used `AllowList` to provide permission control to the `HelloWorld` precompile. You can find more information about the AllowList interface [here](#).

Tutorial

Overview

This is a brief overview of what this tutorial will cover.

- Write a Solidity interface
- Generate the precompile template
- Implement the precompile functions in Golang
- Write and run tests

Stateful precompiles are [alpha software](#). Build at your own risk.

In this tutorial, we used a branch based on Subnet-EVM version `v0.5.1`. You can find the branch [here](#). The code in this branch is the same as Subnet-EVM except for the `precompile/contracts/helloworld` directory. The directory contains the code for the `HelloWorld` precompile. We will be using this precompile as an example to learn how to write a stateful precompile. The code in this branch can become outdated. You should always use the latest version of Subnet-EVM when you develop your own precompile.

Prerequisites

This tutorial assumes familiarity with Golang and JavaScript.

Additionally, users should be deeply familiar with the EVM in order to understand its invariants since adding a Stateful Precompile modifies the EVM itself.

Here are some recommended resources to learn the ins and outs of the EVM:

- [The Ethereum Virtual Machine](#)
- [Precompiles in Solidity](#)
- [Deconstructing a Smart Contract](#)
- [Layout of State Variables in Storage](#)
- [Layout in Memory](#)
- [Layout of Call Data](#)
- [Contract ABI Specification](#)

- [Customizing the EVM with Stateful Precompiles](#)

Please install the following before getting started.

First, install the latest version of Go. Follow the instructions [here](#). You can verify by running `go version`.

Set the `$GOPATH` environment variable properly for Go to look for Go Workspaces. Please read [this](#) for details. You can verify by running `echo $GOPATH`.

:::info An easy way to set GOPATH is using the command: `export GOPATH=$HOME/go ...`

As a few things will be installed into `$GOPATH/bin`, please make sure that `$GOPATH/bin` is in your `$PATH`, otherwise, you may get an error running the commands below. To do that, run the command: `export PATH=$PATH:$GOROOT/bin:$GOPATH/bin`

Download the following prerequisites into your `$GOPATH`:

- Git Clone the [Subnet-EVM](#) repository
- Git Clone [AvalancheGo](#) repository
- Install [Avalanche Network Runner](#)
- Install [solc](#)
- Install [yarn](#)

For easy copy-paste, use the below commands:

```
cd $GOPATH
mkdir -p src/github.com/ava-labs
cd src/github.com/ava-labs
git clone https://github.com/ava-labs/subnet-evm.git
git clone https://github.com/ava-labs/avalanchego.git
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-network-runner/main/scripts/install.sh | sh -
npm install -g solc
npm install -g yarn
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/subnet-evm.git
git clone git@github.com:ava-labs/avalanchego.git
```

You can find more about SSH and how to use it [here](#). :::

Complete Code

You can inspect the [Hello World Pull Request](#) for the complete code.

For a full-fledged example, you can also check out the [Reward Manager Precompile](#)

Step 0: Generating the Precompile

For the tutorial, we will be working in a new branch in Subnet-EVM:

```
cd $GOPATH/src/github.com/ava-labs/subnet-evm
git checkout -b hello-world-stateful-precompile
```

We will start off in this directory `./contract-examples/contracts`:

```
cd contract-examples/contracts
```

First, we must create the Solidity interface that we want our precompile to implement. This will be the `HelloWorld` Interface. It will have two simple functions, `sayHello()` and `setGreeting()`. These two functions will demonstrate the getting and setting respectively of a value stored in the precompile's state space.

Create a new file called `IHelloWorld.sol` and copy and paste the below code:

```
// (c) 2022-2023, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;
import "./IAccount.sol";

interface IHelloWorld is IAllowList {
    // sayHello returns the stored greeting string
    function sayHello() external view returns (string calldata result);

    // setGreeting stores the greeting string
    function setGreeting(string calldata response) external;
}
```

This interface defines 2 functions, `sayHello()` and `setGreeting()`. The `sayHello()` function is a `view` function, meaning it does not modify the state of the precompile and returns a string result. The `setGreeting()` function is a state changer function, meaning it modifies the state of the precompile.

IAllowList is an interface that we will use to restrict access to the precompile. It is defined in `./contract-examples/contracts/IAllowList.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IAllowList {
    // Set [addr] to have the admin role over the precompile contract.
    function setAdmin(address addr) external;

    // Set [addr] to be enabled on the precompile contract.
    function setEnabled(address addr) external;

    // Set [addr] to have no role for the precompile contract.
    function setNone(address addr) external;

    // Read the status of [addr].
    function readAllowList(address addr) external view returns (uint256 role);
}
```

Now we have an interface that our precompile can implement! Let's create an [ABI](#) of our Solidity interface.

In the same `./contract-examples/contracts` directory, let's run

```
solcjs --abi IHelloWorld.sol && mv IHelloWorld_sol_IHelloWorld.abi IHelloWorld.abi
```

This generates the ABI code and moves it to `./contract-examples/contracts` as `IHelloWorld.abi`:

```
[  
 {  
     "inputs": [  
         { "internalType": "address", "name": "addr", "type": "address" }  
     ],  
     "name": "readAllowList",  
     "outputs": [  
         { "internalType": "uint256", "name": "role", "type": "uint256" }  
     ],  
     "stateMutability": "view",  
     "type": "function"  
 },  
 {  
     "inputs": [],  
     "name": "sayHello",  
     "outputs": [  
         { "internalType": "string", "name": "result", "type": "string" }  
     ],  
     "stateMutability": "view",  
     "type": "function"  
 },  
 {  
     "inputs": [  
         { "internalType": "address", "name": "addr", "type": "address" }  
     ],  
     "name": "setAdmin",  
     "outputs": [],  
     "stateMutability": "nonpayable",  
     "type": "function"  
 },  
 {  
     "inputs": [  
         { "internalType": "address", "name": "addr", "type": "address" }  
     ],  
     "name": "setEnabled",  
     "outputs": [],  
     "stateMutability": "nonpayable",  
     "type": "function"  
 },  
 {  
     "inputs": [  
         { "internalType": "string", "name": "response", "type": "string" }  
     ],  
     "name": "setGreeting",  
     "outputs": [],  
     "stateMutability": "nonpayable",  
     "type": "function"  
 },  
 {  
     "inputs": [  
         { "internalType": "address", "name": "addr", "type": "address" }  
     ],  
     "name": "setNone",  
     "outputs": [],  
     "stateMutability": "nonpayable",  
     "type": "function"  
 }]
```

```

        {
            "internalType": "address",
            "name": "addr",
            "type": "address"
        }
    ],
    "name": "setNone",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
}
]

```

As you can see the ABI also contains the `IAllowList` interface functions. This is because the `IHelloWorld` interface inherits from the `IAllowList` interface.

Note: The ABI must have named outputs in order to generate the precompile template.

Now that we have an ABI for the precompile gen tool to interact with, we can run the following command to generate our `HelloWorld` precompile files!

Let's go back to the root of the Subnet-EVM repository and run the `PrecompileGen` script helper:

```

$ cd $GOPATH/src/github.com/ava-labs/subnet-evm
$ ./scripts/generate_precompile.sh --help

Using branch: precompile-tutorial
NAME:
precompilegen - subnet-evm precompile generator tool

USAGE:
main [global options] command [command options] [arguments...]

VERSION:
1.10.26-stable

COMMANDS:
help, h Shows a list of commands or help for one command

GLOBAL OPTIONS:

--abi value
    Path to the contract ABI json to generate, - for STDIN

--out value
    Output folder for the generated precompile files, - for STDOUT (default =
    ./precompile/contracts/{pkg}). Test files won't be generated if STDOUT is used

--pkg value
    Go package name to generate the precompile into (default = {type})

--type value
    Struct name for the precompile (default = {abi file name})

MISC

--help, -h          (default: false)
    show help

--version, -v       (default: false)
    print the version

COPYRIGHT:
Copyright 2013-2022 The go-ethereum Authors

```

Precompile contracts reside under the `./precompile/contracts` directory. Let's generate our precompile template in the `./precompile/contracts/helloworld` directory, where `helloworld` is the name of the Go package we want to generate the precompile into.

```
./scripts/generate_precompile.sh --abi ./contract-examples/contracts/IHelloWorld.abi --type HelloWorld --pkg helloworld
```

This generates a precompile template files `contract.go`, `contract.abi`, `config.go`, `module.go` and `README.md` located at `./precompile/contracts/helloworld` directory. `README.md` explains general guidelines for precompile development. You should carefully read this file before modifying the precompile template.

There are some must-be-done changes waiting in the generated file. Each area requiring you to add your code is marked with CUSTOM CODE to make them easy to find and modify.
Additionally there are other files you need to edit to activate your precompile.
These areas are highlighted with comments "ADD YOUR PRECOMPILE HERE".
For testing take a look at other precompile tests in `contract_test.go` and `config_test.go` in other precompile folders.
See the tutorial in <<https://docs.avax.network/subnets/hello-world-precompile-tutorial>> for more information about precompile

development.

General guidelines for precompile development:

- 1- Set a suitable config key in generated module.go. E.g: "yourPrecompileConfig"
- 2- Read the comment and set a suitable contract address in generated module.go. E.g:
ContractAddress = common.HexToAddress("ASUITABLEHEXADDRESS")
- 3- It is recommended to only modify code in the highlighted areas marked with "CUSTOM CODE STARTS HERE". Typically, custom codes are required in only those areas.
Modifying code outside of these areas should be done with caution and with a deep understanding of how these changes may impact the EVM.
- 4- Set gas costs in generated contract.go
- 5- Force import your precompile package in precompile/registry/registry.go
- 6- Add your config unit tests under generated package config_test.go
- 7- Add your contract unit tests under generated package contract_test.go
- 8- Additionally you can add a full-fledged VM test for your precompile under plugin/vm/vm_test.go. See existing precompile tests for examples.
- 9- Add your solidity interface and test contract to contract-examples/contracts
- 10- Write solidity tests for your precompile in contract-examples/test
- 11- Create your genesis with your precompile enabled in tests/precompile/genesis/
- 12- Create e2e test for your solidity test in tests/precompile/solidity/suites.go
- 13- Run your e2e precompile Solidity tests with './scripts/run_ginkgo.sh'

Let's follow these steps and create our HelloWorld precompile!

Step 1: Set Config Key

Let's jump to `helloworld/module.go` file first. This file contains the module definition for our precompile. You can see the `ConfigKey` is set to some default value of `helloWorldConfig`. This key should be unique to the precompile. This config key determines which JSON key to use when reading the precompile's config from the JSON upgrade/genesis file. In this case, the config key is `helloWorldConfig` and the JSON config should look like this:

```
{  
    "helloWorldConfig": {  
        "blockTimestamp": 0  
        ...  
    }  
}
```

Step 2: Set Contract Address

You can see the `ContractAddress` is set to some default value. This should be changed to a suitable address for your precompile. The address should be unique to the precompile. There is a registry of precompile addresses under [precompile/registry/registry.go](#). A list of addresses is specified in the comments under this file. Modify the default value to be the next user available stateful precompile address. For forks of Subnet-EVM, users should start at `0x0300` to ensure that their own modifications do not conflict with stateful precompiles that may be added to Subnet-EVM in the future. You should pick an address that is not already taken, and write it down in `registry.go` as a comment for future reference.

```

// This list is kept just for reference. The actual addresses defined in respective packages of precompiles.
// Note: it is important that none of these addresses conflict with each other or any other precompiles
// in core/vm/contracts.go.
// The first stateful precompiles were added in coreth to support nativeAssetCall and nativeAssetBalance. New stateful
precompiles
// originating in coreth will continue at this prefix, so we reserve this range in subnet-evm so that they can be migrated into
// subnet-evm without issue.
// These start at the address: 0x01000000000000000000000000000000 and will increment by 1.
// Optional precompiles implemented in subnet-evm start at 0x02000000000000000000000000000000 and will increment by 1
// from here to reduce the risk of conflicts.
// For forks of subnet-evm, users should start at 0x03000000000000000000000000000000 to ensure
// that their own modifications do not conflict with stateful precompiles that may be added to subnet-evm
// in the future.
// ContractDeployerAllowListAddress = common.HexToAddress("0x02000000000000000000000000000000")
// ContractNativeMinterAddress = common.HexToAddress("0x02000000000000000000000000000001")
// TxAllowListAddress = common.HexToAddress("0x02000000000000000000000000000002")
// FeeManagerAddress = common.HexToAddress("0x02000000000000000000000000000003")
// RewardManagerAddress = common.HexToAddress("0x02000000000000000000000000000004")
// HelloWorldAddress = common.HexToAddress("0x03000000000000000000000000000000")
// ADD YOUR PRECOMPILE HERE
// {YourPrecompile}Address = common.HexToAddress("0x03000000000000000000000000000000??"")

```

Don't forget to update the actual variable `ContractAddress` in `module.go` to the address you chose. It should look like this:

Now when Subnet-EVM sees the `helloworld.ContractAddress` as input when executing `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`, it can run the precompile if the precompile is enabled.

Step 3: Add Custom Code

Search (`CTRL F`) throughout the file with `CUSTOM CODE STARTS HERE` to find the areas in the precompile package that you need to modify. You should start with the reference imports code block.

Step 3.1: Module File

The module file contains fundamental information about the precompile. This includes the key for the precompile, the address of the precompile, and a configurator. This file is located at [`./precompile/helloworld/module.go`](#).

This file defines the module for the precompile. The module is used to register the precompile to the precompile registry. The precompile registry is used to read configs and enable the precompile. Registration is done in the `init()` function of the module file. `MakeConfig()` is used to create a new instance for the precompile config. This will be used in custom Unmarshal/Marshal logic. You don't need to override these functions.

Configure()

Module file contains a `configurator` which implements the `contract.Configurator` interface. This interface includes a `Configure()` function used to configure the precompile and set the initial state of the precompile. This function is called when the precompile is enabled. This is typically used to read from a given config in upgrade/genesis JSON and sets the initial state of the precompile accordingly. This function also calls `AllowListConfig.Configure()` to invoke AllowList configuration as the last step. You should keep it as is if you want to use AllowList. You can modify this function for your custom logic. You can circle back to this function later after you have finalized the implementation of the precompile config.

Step 3.2: Config File

The config file contains the config for the precompile. This file is located at [`./precompile/helloworld/config.go`](#). This file contains the `Config` struct, which implements `precompileconfig.Config` interface. It has some embedded structs like `precompileconfig.Upgrade`. `Upgrade` is used to enable upgrades for the precompile. It contains the `BlockTimestamp` and `Disable` to enable/disable upgrades. `BlockTimestamp` is the timestamp of the block when the upgrade will be activated. `Disable` is used to disable the upgrade. If you use `AllowList` for the precompile, there is also `allowlist.AllowListConfig` embedded in the `Config` struct. `AllowListConfig` is used to specify initial roles for specified addresses. If you have any custom fields in your precompile config, you can add them here. These custom fields will be read from upgrade/genesis JSON and set in the precompile config.

```
// Config implements the precompileconfig.Config interface and
// adds specific configuration for HelloWorld.
type Config struct {
    allowlist.AllowListConfig
    precompileconfig.Upgrade
}
```

Verify()

`Verify()` is called on startup and an error is treated as fatal. Generated code contains a call to `AllowListConfig.Verify()` to verify the `AllowListConfig`. You can leave that as is and start adding your own custom verify code after that.

We can leave this function as is right now because there is no invalid custom configuration for the `Config`.

```
// Verify tries to verify Config and returns an error accordingly.
func (c *Config) Verify() error {
    // Verify AllowList first
    if err := c.AllowListConfig.Verify(); err != nil {
        return err
    }

    // CUSTOM CODE STARTS HERE
    // Add your own custom verify code for Config here
    // and return an error accordingly
    return nil
}
```

Equal()

Next, we see is `Equal()`. This function determines if two precompile configs are equal. This is used to determine if the precompile needs to be upgraded. There is some default code that is generated for checking `Upgrade` and `AllowListConfig` equality.

```
// Equal returns true if [s] is a [*Config] and it has been configured identical to [c].
func (c *Config) Equal(s precompileconfig.Config) bool {
    // typecast before comparison
    other, ok := (s).(*Config)
    if !ok {
        return false
    }

    // CUSTOM CODE STARTS HERE
    // modify this boolean accordingly with your custom Config, to check if [other] and the current [c] are equal
    // if Config contains only Upgrade and AllowListConfig you can skip modifying it.
    equals := c.Upgrade.Equal(&other.Upgrade) && c.AllowListConfig.Equal(&other.AllowListConfig)
    return equals
}
```

We can leave this function as is since we check `Upgrade` and `AllowListConfig` for equality which are the only fields that `Config` struct has.

Step 3.3: Modify Configure()

We can now circle back to `Configure()` in `module.go` as we finished implementing `Config` struct. This function configures the `state` with the initial configuration at `blockTimestamp` when the precompile is enabled. In the `HelloWorld` example, we want to set up a default key-value mapping in the state where the key is `storageKey` and the value is `Hello World!`. The `StateDB` allows us to store a key-value mapping of 32-byte hashes. The below code snippet can be copied and pasted to overwrite the default `Configure()` code.

```
const defaultGreeting = "Hello World!"

// Configure configures [state] with the given [cfg] precompileconfig.
// This function is called by the EVM once per precompile contract activation.
// You can use this function to set up your precompile contract's initial state,
// by using the [cfg] config and [state] stateDB.
func (*configurator) Configure(chainConfig contract.ChainConfig, cfg precompileconfig.Config, state contract.StateDB, _ contract.BlockContext) error {
    config, ok := cfg.(*Config)
    if !ok {
        return fmt.Errorf("incorrect config %T: %v", config, config)
    }
    // CUSTOM CODE STARTS HERE

    // This will be called in the first block where HelloWorld stateful precompile is enabled.
    // 1) If BlockTimestamp is nil, this will not be called
    // 2) If BlockTimestamp is 0, this will be called while setting up the genesis block
    // 3) If BlockTimestamp is 1000, this will be called while processing the first block
    // whose timestamp is >= 1000
    //
    // Set the initial value under [common.BytesToHash([]byte("storageKey"))] to "Hello World!"
    StoreGreeting(state, defaultGreeting)
    // AllowList is activated for this precompile. Configuring allowlist addresses here.
    return config.AllowListConfig.Configure(state, ContractAddress)
}
```

Step 3.4: Contract File

The contract file contains the functions of the precompile contract that will be called by the EVM. The file is located at

`./precompile/helloworld/contract.go`. Since we use `IAccount` interface there will be auto-generated code for `AllowList` functions like below:

```
// GetHelloWorldAllowListStatus returns the role of [address] for the HelloWorld list.
func GetHelloWorldAllowListStatus(stateDB contract.StateDB, address common.Address) allowlist.Role {
    return allowlist.GetAllowListStatus(stateDB, ContractAddress, address)
}

// SetHelloWorldAllowListStatus sets the permissions of [address] to [role] for the
// HelloWorld list. Assumes [role] has already been verified as valid.
// This stores the [role] in the contract storage with address [ContractAddress]
// and [address] hash. It means that any reusage of the [address] key for different value
// conflicts with the same slot [role] is stored.
// Precompile implementations must use a different key than [address] for their storage.
func SetHelloWorldAllowListStatus(stateDB contract.StateDB, address common.Address, role allowlist.Role) {
    allowlist.SetAllowListRole(stateDB, ContractAddress, address, role)
}
```

These will be helpful to use `AllowList` precompile helper in our functions.

Packers and Unpackers

There are also auto-generated Packers and Unpackers for the ABI. These will be used in `sayHello` and `setGreeting` functions to comfort the ABI. These functions are auto-generated and will be used in necessary places accordingly. You don't need to worry about how to deal with them, but it's good to know what they are.

Each input to a precompile contract function has its own `Unpacker` function as follows:

```
// UnpackSetGreetingInput attempts to unpack [input] into the string type argument
// assumes that [input] does not include selector (omits first 4 func signature bytes)
func UnpackSetGreetingInput(input []byte) (string, error) {
    res, err := HelloWorldABI.UnpackInput("setGreeting", input)
    if err != nil {
        return "", err
    }
    unpacked := *abi.ConvertType(res[0], new(string)).(*string)
    return unpacked, nil
}
```

The ABI is a binary format and the input to the precompile contract function is a byte array. The `Unpacker` function converts this input to a more easy-to-use format so that we can use it in our function.

Similarly, there is a `Packer` function for each output of a precompile contract function as follows:

```
// PackSayHelloOutput attempts to pack given result of type string
// to conform the ABI outputs.
func PackSayHelloOutput(result string) ([]byte, error) {
    return HelloWorldABI.PackOutput("sayHello", result)
}
```

This function converts the output of the function to a byte array that conforms to the ABI and can be returned to the EVM as a result.

Modify sayHello()

The next place to modify is in our `sayHello()` function. In a previous step, we created the `IHelloWorld.sol` interface with two functions `sayHello()` and `setGreeting()`. We finally get to implement them here. If any contract calls these functions from the interface, the below function gets executed. This function is a simple getter function. In `Configure()` we set up a mapping with the key as `storageKey` and the value as `Hello World!`. In this function, we will be returning whatever value is at `storageKey`. The below code snippet can be copied and pasted to overwrite the default `setGreeting` code.

First, we add a helper function to get the greeting value from the stateDB, this will be helpful when we test our contract.

```
// GetGreeting returns the value of the storage key "storageKey" in the contract storage,
// with leading zeroes trimmed.
// This function is mostly used for tests.
func GetGreeting(stateDB contract.StateDB) string {
    // Get the value set at recipient
    value := stateDB.GetState(ContractAddress, storageKeyHash)
    return string(common.TrimLeftZeroes(value.Bytes()))
}
```

Now we can modify the `sayHello` function to return the stored value.

```
func sayHello(accessibleState contract.AccessibleState, caller common.Address, addr common.Address, input []byte, suppliedGas
uint64, readOnly bool) (ret []byte, remainingGas uint64, err error) {
    if remainingGas, err = contract.deductGas(suppliedGas, SayHelloGasCost); err != nil {
        return nil, 0, err
    }
    // CUSTOM CODE STARTS HERE

    // Get the current state
    currentState := accessibleState.GetStateDB()
    // Get the value set at recipient
    value := GetGreeting(currentState)
    packedOutput, err := PackSayHelloOutput(value)
    if err != nil {
        return nil, remainingGas, err
    }

    // Return the packed output and the remaining gas
    return packedOutput, remainingGas, nil
}
```

Modify setGreeting()

We can also modify our `setGreeting()` function. This is a simple setter function. It takes in `input` and we will set that as the value in the state mapping with the key as `storageKey`. It also checks if the VM running the precompile is in read-only mode. If it is, it returns an error.

There is also a generated `AllowList` code in that function. This generated code checks if the caller address is eligible to perform this state-changing operation. If not, it returns an error.

Let's add the helper function to set the greeting value in the stateDB, this will be helpful when we test our contract.

```
// StoreGreeting sets the value of the storage key "storageKey" in the contract storage.
func StoreGreeting(stateDB contract.StateDB, input string) {
    inputPadded := common.LeftPadBytes([]byte(input), common.HashLength)
    inputHash := common.BytesToHash(inputPadded)

    stateDB.SetState(ContractAddress, storageKeyHash, inputHash)
}
```

The below code snippet can be copied and pasted to overwrite the default `setGreeting()` code.

```
func setGreeting(accessibleState contract.AccessibleState, caller common.Address, addr common.Address, input []byte, suppliedGas
uint64, readOnly bool) (ret []byte, remainingGas uint64, err error) {
    if remainingGas, err = contract.DeductGas(suppliedGas, SetGreetingGasCost); err != nil {
        return nil, 0, err
    }
    if readOnly {
        return nil, remainingGas, vmerrs.ErrWriteProtection
    }
```

```

}

// attempts to unpack [input] into the arguments to the SetGreetingInput.
// Assumes that [input] does not include selector
// You can use unpacked [inputStruct] variable in your code
inputStruct, err := UnpackSetGreetingInput(input)
if err != nil {
    return nil, remainingGas, err
}

// Allow list is enabled and SetGreeting is a state-changer function.
// This part of the code restricts the function to be called only by enabled/admin addresses in the allow list.
// You can modify/delete this code if you don't want this function to be restricted by the allow list.
stateDB := accessibleState.GetStateDB()
// Verify that the caller is in the allow list and therefore has the right to call this function.
callerStatus := allowlist.GetAllowListStatus(stateDB, ContractAddress, caller)
if !callerStatus.IsEnabled() {
    return nil, remainingGas, fmt.Errorf("%w: %s", ErrCannotSetGreeting, caller)
}
// allow list code ends here.

// CUSTOM CODE STARTS HERE
// Check if the input string is longer than HashLength
if len(inputStruct) > common.HashLength {
    return nil, 0, ErrInputExceedsLimit
}

// setGreeting is the execution function
// "SetGreeting(name string)" and sets the storageKey
// in the string returned by hello world
StoreGreeting(stateDB, inputStruct)

// This function does not return an output, leave this one as is
packedOutput := []byte{}

// Return the packed output and the remaining gas
return packedOutput, remainingGas, nil
}

```

Step 4: Set Gas Costs

In [precompile/contract/utils.go](#) we have `WriteGasCostPerSlot` and `ReadGasCostPerSlot`. Setting gas costs is very important for these functions, and should be done carefully. If the gas costs are set too low, then these can be abused and can cause Dos attacks. If the gas costs are set too high, then the contract will be too expensive to run. In order to provide a baseline for gas costs, we have set the following gas costs.

```

// Gas costs for stateful precompiles
const {
    WriteGasCostPerSlot = 20_000
    ReadGasCostPerSlot = 5_000
}

```

`WriteGasCostPerSlot` is the cost of one write such as modifying a state storage slot.

`ReadGasCostPerSlot` is the cost of reading a state storage slot.

This should be in your gas cost estimations based on how many times the precompile function does a read or a write. For example, if the precompile modifies the state slot of its precompile address twice then the gas cost for that function would be `40_000`. However, if the precompile does additional operations and requires more computational power, then you should increase the gas costs accordingly.

On top of these gas costs, we also have to account for the gas costs of AllowList gas costs. These are the gas costs of reading and writing permissions for addresses in AllowList. These are defined under [precompile/allowlist/allowlist.go](#). By default, these are added to the default gas costs of the state-change functions (SetGreeting) of the precompile. Meaning that these functions will cost an additional `ReadAllowListGasCost` in order to read permissions from the storage. If you don't plan to read permissions from the storage then you can omit these.

Now going back to [/precompile/helloworld/contract.go](#), we can modify our precompile function gas costs. Please search (`CTRL F`) `SET A GAS COST HERE` to locate the default gas cost code.

```

SayHelloGasCost uint64 = 0 // SET A GAS COST HERE
SetGreetingGasCost uint64 = 0 + allowlist.ReadAllowListGasCost // SET A GAS COST HERE

```

We get and set our greeting with `sayHello()` and `setGreeting()` in one slot respectively so we can define the gas costs as follows. We also read permissions from the AllowList in `setGreeting()` so we keep `allowlist.ReadAllowListGasCost`.

```

SayHelloGasCost uint64 = contract.ReadGasCostPerSlot
SetGreetingGasCost uint64 = contract.WriteGasCostPerSlot + allowlist.ReadAllowListGasCost

```

Step 5: Register Precompile

We should register our precompile in `/precompile/registry.go` to be discovered by other packages. Our `Module` file contains an `init()` function that registers our precompile. `init()` is called when the package is imported. We should register our precompile in a common package so that it can be imported by other packages.

```
// Force imports of each precompile to ensure each precompile's init function runs and registers itself
// with the registry.
import (
    - "github.com/ava-labs/subnet-evm/precompile/contracts/deployerallowlist"
    - "github.com/ava-labs/subnet-evm/precompile/contracts/nativeminter"
    - "github.com/ava-labs/subnet-evm/precompile/contracts/txallowlist"
    - "github.com/ava-labs/subnet-evm/precompile/contracts/feemanager"
    - "github.com/ava-labs/subnet-evm/precompile/contracts/rewardmanager"
    - "github.com/ava-labs/subnet-evm/precompile/contracts/helloworld"
    // ADD YOUR PRECOMPILE HERE
    // - "github.com/ava-labs/subnet-evm/precompile/contracts/yourprecompile"
)
```

Step 6: Add Config Tests

Precompile generation tool generates skeletons for unit tests as well. Generated config tests will be under `/precompile/contracts/helloworld/config_test.go`. There are mainly two functions we need to test: `Verify` and `Equal`. `Verify` checks if the precompile is configured correctly. `Equal` checks if the precompile is equal to another precompile. Generated `Verify` tests contain a valid case. You can add more invalid cases depending on your implementation. `Equal` tests generate some invalid cases to test different timestamps, types, and AllowList cases. You can check other `config_test.go` files in the `/precompile/contracts` directory for more examples. For the `HelloWorld` precompile, you can check the generated code [here](#).

Step 7: Add Contract Tests

The tool also generates contract tests to make sure our precompile is working correctly. Generated tests include cases to test allow list capabilities, gas costs, and calling functions in read-only mode. You can check other `contract_test.go` files in the `/precompile/contracts`. Hello World contract tests will be under `/precompile/contracts/helloworld/contract_test.go` [here](#). We will also add more tests to cover functionalities of `sayHello()` and `setGreeting()`. Contract tests are defined in a standard structure that each test can customize to their needs. The test structure is as follows:

```
// PrecompileTest is a test case for a precompile
type PrecompileTest struct {
    // Caller is the address of the precompile caller
    Caller common.Address
    // Input the raw input bytes to the precompile
    Input []byte
    // InputFn is a function that returns the raw input bytes to the precompile
    // If specified, Input will be ignored.
    InputFn func(t *testing.T) []byte
    // SuppliedGas is the amount of gas supplied to the precompile
    SuppliedGas uint64
    // ReadOnly is whether the precompile should be called in read only
    // mode. If true, the precompile should not modify the state.
    ReadOnly bool
    // Config is the config to use for the precompile
    // It should be the same precompile config that is used in the
    // precompile's configurator.
    // If nil, Configure will not be called.
    Config precompileconfig.Config
    // BeforeHook is called before the precompile is called.
    BeforeHook func(t *testing.T, state contract.StateDB)
    // AfterHook is called after the precompile is called.
    AfterHook func(t *testing.T, state contract.StateDB)
    // ExpectedRes is the expected raw byte result returned by the precompile
    ExpectedRes []byte
    // ExpectedErr is the expected error returned by the precompile
    ExpectedErr string
    // BlockNumber is the block number to use for the precompile's block context
    BlockNumber int64
}
```

Each test can populate the fields of the `PrecompileTest` struct to customize the test. This test uses an `AllowList` helper function `allowlist.RunPrecompileWithAllowListTests(t, Module, state.NewTestStateDB, tests)` which can run all specified tests plus `AllowList` test suites. If you don't plan to use `AllowList`, you can directly run them as follows:

```
for name, test := range tests {
    t.Run(name, func(t *testing.T) {
        test.Run(t, module, newStateDB(t))
```

```
    })
}
```

Step 8 (Optional): VM Tests

VM tests are tests that run the precompile by calling it through the Subnet-EVM. These are the most comprehensive tests that we can run. If your precompile modifies how the Subnet-EVM works, for example changing blockchain rules, you should add a VM test. For example, you can take a look at the `TestRewardManagerPrecompileSetRewardAddress` function in [here](#). For this Hello World example, we don't modify any Subnet-EVM rules, so we don't need to add any VM tests.

Step 9: Add Test Contract

Let's add our test contract to `./contract-examples/contracts`. This smart contract lets us interact with our precompile! We cast the `HelloWorld` precompile address to the `IHelloWorld` interface. In doing so, `helloWorld` is now a contract of type `IHelloWorld`, and when we call any functions on that contract, we will be redirected to the `HelloWorld` precompile address. The below code snippet can be copied and pasted into a new file called `ExampleHelloWorld.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./IHelloWorld.sol";

// ExampleHelloWorld shows how the HelloWorld precompile can be used in a smart contract.
contract ExampleHelloWorld {
    address constant HELLO_WORLD_ADDRESS = 0x030000000000000000000000000000000000000000000000000000000000000;
    IHelloWorld helloWorld = IHelloWorld(HELLO_WORLD_ADDRESS);

    function getHello() public view returns (string memory) {
        return helloWorld.sayHello();
    }

    function setGreeting(string calldata greeting) public {
        helloWorld.setGreeting(greeting);
    }
}
```

:::warning

Hello World Precompile is a different contract than `ExampleHelloWorld` and has a different address. Since the precompile uses `AllowList` for a permissioned access, any call to the precompile including from `ExampleHelloWorld` will be denied unless the caller is added to the `AllowList`.

:::

Please note that this contract is simply a wrapper and is calling the precompile functions.

Step 10: Add Hardhat Test

We can now write our hardhat test in `./contract-examples/test`. The below code snippet can be copied and pasted into a new file called `hello_world.ts` [here](#):

```
// (c) 2019-2022, Ava Labs, Inc. All rights reserved.
// See the file LICENSE for licensing terms.

import { expect } from "chai";
import { ethers } from "hardhat";
import { SignerWithAddress } from "@nomiclabs/hardhat-ethers/signers";
import { Contract, ContractFactory } from "ethers";

// make sure this is always an admin for the precompile
const adminAddress: string = "0x8db97c7eC249c2b98bDC0226Cc4C2A57BF52FC";
const HELLO_WORLD_ADDRESS = "0x030000000000000000000000000000000000000000000000000000000000000";

describe("ExampleHelloWorld", function () {
    let helloWorldContract: Contract;
    let adminSigner: SignerWithAddress;
    let adminSignerPrecompile: Contract;

    before(async function () {
        // Deploy Hello World Contract
        const ContractF: ContractFactory = await ethers.getContractFactory(
            "ExampleHelloWorld"
        );
        helloWorldContract = await ContractF.deploy();
        await helloWorldContract.deployed();
        const helloWorldContractAddress: string = helloWorldContract.address;
        console.log(`Contract deployed to: ${helloWorldContractAddress}`);

        adminSigner = await ethers.getSigner(adminAddress);
    });
})
```

```

adminSignerPrecompile = await ethers.getContractAt(
  "IHelloWorld",
  HELLO_WORLD_ADDRESS,
  adminSigner
);
});

it("should getHello properly", async function () {
  let result = await helloWorldContract.callStatic.getHello();
  expect(result).to.equal("Hello World!");
});

it("contract should not be able to set greeting without enabled", async function () {
  const modifiedGreeting = "What's up";
  let contractRole = await adminSignerPrecompile.readAllowList(
    helloWorldContract.address
  );
  expect(contractRole).to.be.equal(0); // 0 = NONE
  try {
    let tx = await helloWorldContract.setGreeting(modifiedGreeting);
    await tx.wait();
  } catch (err) {
    return;
  }
  expect.fail("should have errored");
});

it("should be add contract to enabled list", async function () {
  let contractRole = await adminSignerPrecompile.readAllowList(
    helloWorldContract.address
  );
  expect(contractRole).to.be.equal(0);

  let enableTx = await adminSignerPrecompile.setEnabled(
    helloWorldContract.address
  );
  await enableTx.wait();
  contractRole = await adminSignerPrecompile.readAllowList(
    helloWorldContract.address
  );
  expect(contractRole).to.be.equal(1); // 1 = ENABLED
});

it("should setGreeting and getHello", async function () {
  const modifiedGreeting = "What's up";
  let tx = await helloWorldContract.setGreeting(modifiedGreeting);
  await tx.wait();

  expect(await helloWorldContract.callStatic.getHello()).to.be.equal(
    modifiedGreeting
  );
});
});

```

Step 11: Add Genesis

To run our HardHat test, we will need to create a Subnet that has the `Hello World` precompile activated, so we will copy and paste the below genesis file into: `./tests/precompile/genesis/hello_world.json`.

Note: it's important that this has the same name as the HardHat test file we created in Step 8.1.

```
{
  "config": {
    "chainId": 99999,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip150Hash": "0x2086799aeebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "istanbulBlock": 0,
    "muirGlacierBlock": 0,
    "subnetEVMTimestamp": 0,
    "feeConfig": {
      "gasLimit": 20000000,
      "minBaseFee": 1000000000,
    }
  }
}
```

Adding this to our genesis enables our HelloWorld precompile at the very first block (timestamp 0), with `0x8db97c7cEcE249c2b98bd0226Cc4C2A57BF52FC` as the admin address.

```
{  
    "helloWorldConfig": {  
        "blockTimestamp": 0,  
        "adminAddresses": ["0x8db97C7cEcE249c2b98bdC0226Cc4C2A57BF52FC"]  
    }  
}
```

Step 12: Declaring the HardHat E2E Test

Now that we have declared the HardHat test and corresponding `genesis.json` file. The last step to running the e2e test is to declare the new test in `./tests/precompile/solidity/suites.go`.

At the bottom of the file you will see the following code commented out:

```
// TODO: can we refactor this so that it automagically checks to ensure each hardhat test file matches the name of a hardhat
genesis file
// and then runs the hardhat tests for each one without forcing precompile developers to modify this file.
// ADD YOUR PRECOMPILE HERE
/*
    ginkgo.It("your precompile", ginkgo.Label("Precompile"), ginkgo.Label("YourPrecompile"), func() {
        ctx, cancel := context.WithTimeout(context.Background(), time.Minute)
        defer cancel()

        // Specify the name shared by the genesis file in ./tests/precompile/genesis/{your_precompile}.json
        // and the test file in ./contract-examples/tests/{your_precompile}.ts
        utils.ExecuteHardHatTestOnNewBlockchain(ctx, "your_precompile")
    })
*/
```

You should copy and paste the ginkgo It node and update from `{your_precompile}` to `hello_world`. The string passed into `utils.ExecuteHardHatTestsOnNewBlockchain(ctx, "your_precompile")` will be used to find both the HardHat test file to execute and the genesis file, which is why you need to use the same name for both.

After modifying the `TT` node, it should look like the following (you can copy and paste this directly if you prefer):

```
ginkgo.It("hello world", ginkgo.Label("Precompile"), ginkgo.Label("HelloWorld"), func() {
    ctx, cancel := context.WithTimeout(context.Background(), time.Minute)
    defer cancel()

    utils.ExecuteHardHatTestOnNewBlockchain(ctx, "hello_world")
})
```

Now that we've set up the new ginkgo test, we can run the ginkgo test that we want by using the `GINKO_LABEL_FILTER`. This environment variable is passed as a flag to Ginkgo in `./scripts/run_ginkgo.sh` and restricts what tests will run to only the tests with a matching label.

Step 13: Running E2E Tests

Building AvalancheGo and Subnet-EVM

Before we start testing, we will need to build the AvalancheGo binary and the custom Subnet-EVM binary.

You should have cloned [AvalancheGo](#) within your `$GOPATH` in the [Prerequisites](#) section, so you can build AvalancheGo with the following command:

```
cd $GOPATH/src/github.com/ava-labs/avalanchego  
./scripts/build.sh
```

Once you've built AvalancheGo, you can confirm that it was successful by printing the version:

```
./build/avalanchego --version
```

This should print something like the following (if you are running AvalancheGo v1.9.7):

```
avalanche/1.9.7 [database=v1.4.5, rpcchainvm=22, commit=3e3e40f2f4658183d999807b724245023a13f5dc]
```

This path will be used later as the environment variable `AVALANCHEGO_EXEC_PATH` in the network runner.

Please note that the RPCChainVM version of AvalancheGo and Subnet-EVM must match.

Once we've built AvalancheGo, we can navigate back to the Subnet-EVM repo and build the Subnet-EVM binary:

```
cd $GOPATH/src/github.com/ava-labs/subnet-evm  
./scripts/build.sh
```

This will build the Subnet-EVM binary and place it in AvalancheGo's `build/plugins` directory by default at the file path:

```
$GOPATH/src/github.com/ava-labs/avalanchego/build/plugins/srEXiWaHuhNyGwPUi444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy
```

To confirm that the Subnet-EVM binary is compatible with AvalancheGo, you can run the same version command and confirm the RPCChainVM version matches:

```
$GOPATH/src/github.com/ava-labs/avalanchego/build/plugins/srEXiWaHuhNyGwPUi444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy --version
```

This should give similar output:

```
Subnet-EVM/v0.4.10@a584fcad593885b6c095f42adaff6b53d51aedb8 [AvalancheGo=v1.9.7, rpcchainvm=22]
```

If the RPCChainVM Protocol version printed out does not match the one used in AvalancheGo then Subnet-EVM will not be able to talk to AvalancheGo and the blockchain will not start.

The `build/plugins` directory will later be used as the `AVALANCHEGO_PLUGIN_PATH`.

Running Ginkgo Tests

To run ONLY the HelloWorld precompile test, run the command:

```
cd $GOPATH/src/github.com/ava-labs/subnet-evm  
GINKO_LABEL_FILTER=HelloWorld ./scripts/run_ginkgo.sh
```

You will first see the node starting up in the `BeforeSuite` section of the precompile test:

```
$ GINKGO_LABEL_FILTER=HelloWorld ./scripts/run_ginkgo.sh  
Using branch: hello-world-tutorial-walkthrough  
building precompile.test  
# github.com/ava-labs/subnet-evm/tests/precompile.test  
ld: warning: could not create compact unwind for _blst_sha256_block_data_order: does not use RBP or RSP based frame  
  
Compiled precompile.test  
# github.com/ava-labs/subnet-evm/tests/load.test  
ld: warning: could not create compact unwind for _blst_sha256_block_data_order: does not use RBP or RSP based frame  
  
Compiled load.test  
Running Suite: subnet-evm precompile ginkgo test suite - /Users/avalabs/go/src/github.com/ava-labs/subnet-evm  
=====  
Random Seed: 1674833631  
  
Will run 1 of 7 specs  
-----  
[BeforeSuite]  
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/precompile_test.go:31
```

```

> Enter [BeforeSuite] TOP-LEVEL - /Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/precompile_test.go:31
@ 01/27/23 10:33:51.001
INFO [01-27|10:33:51.002] Starting AvalancheGo node          wd=/Users/avalabs/go/src/github.com/ava-labs/subnet-evm
INFO [01-27|10:33:51.002] Executing                         cmd="../scripts/run.sh"
[streaming output] Using branch: hello-world-tutorial-walkthrough
[streaming output] Creating data directory: /tmp/subnet-evm-start-node/2023-01-27:10:33:51/node1
[streaming output] Executing command: /Users/avalabs/go/src/github.com/ava-labs/avalanchego/build/avalanchego --data-dir=/tmp/subnet-evm-start-node/2023-01-27:10:33:51/node1 --config-file=/tmp/subnet-evm-start-node/2023-01-27:10:33:51/node1/config.json
[streaming output] [01-27|10:33:56.962] WARN process/process.go:92 UPnP or NAT-PMP router attach failed, you may not be listening publicly. Please confirm the settings in your router
[streaming output] [01-27|10:33:56.965] INFO leveldb/db.go:203 creating leveldb {"config": {"blockCacheCapacity":12582912, "blockSize":10, "compactionExpandLimitFactor":0, "compactionGPOverlapsFactor":0, "compactionL0Trigger":0}}
[streaming output] [01-27|10:33:57.061] INFO node/node.go:224 initializing networking {"currentNodeIP": "10.56.134.240:9651"}
[streaming output] [01-27|10:33:57.066] INFO server/server.go:309 adding route {"url": "/ext/vm/mgj78GNP7uDwBCCq6YwThhaN8FLyybkCa4zBWTQbNgmK6k9A6", "endpoint": "/rpc"}
[streaming output] [01-27|10:33:57.782] INFO <P Chain> platformvm/vm.go:228 initializing last accepted {"blkID": "2cC67R6vPRSX4BCAY3ouAk9JKCCpAxjFxRVUQWfnEQF1BjQhqX"}
[streaming output] INFO [01-27|10:33:57.792] <C Chain> github.com/ava-labs/coreth/core/state/snapshot/wipe.go:133: Deleted state snapshot leftovers kind=accounts wiped=0 elapsed="26.846μs"
[streaming output] [01-27|10:33:57.801] INFO server/server.go:270 adding route {"url": "/ext/bc/2CA6j5zYzasyNPsFeNoqWkmTCt3VScMvXUZHbfDJ8k3oGzAPTU", "endpoint": "/ws"}
[streaming output] [01-27|10:33:57.806] INFO <P Chain> snowman/transitive.go:444 consensus starting {"lastAcceptedBlock": "2cC67R6vPRSX4BCAY3ouAk9JKCCpAxjFxRVUQWfnEQF1BjQhqX"}
[streaming output] [01-27|10:34:01.004] WARN health/health.go:85 failing readiness check {"reason": {"bootstrapped": {"error": "not yet run", "timestamp": "0001-01-01T00:00:00Z", "duration": "0"}}}
INFO [01-27|10:34:06.003] AvalancheGo node is healthy
< Exit [BeforeSuite] TOP-LEVEL - /Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/precompile_test.go:31 @
01/27/23 10:34:06.003 (15.002s)
[BeforeSuite] PASSED [15.002 seconds]

```

After the `BeforeSuite` completes successfully, it will skip all but the `HelloWorld` labeled precompile test:

```

S [SKIPPED]
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
contract native minter [Precompile, ContractNativeMinter]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:29
-----
S [SKIPPED]
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
tx allow list [Precompile, TxAllowList]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:36
-----
S [SKIPPED]
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
contract deployer allow list [Precompile, ContractDeployerAllowList]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:43
-----
S [SKIPPED]
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
fee manager [Precompile, FeeManager]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:50
-----
S [SKIPPED]
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
reward manager [Precompile, RewardManager]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:57
-----
[Precompiles]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:26
hello world [Precompile, HelloWorld]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:64
> Enter [It] hello world - /Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/precompile/solidity/suites.go:64 @
01/27/23 10:34:06.004
INFO [01-27|10:34:06.004] Executing HardHat tests on a new blockchain test=hello_world
INFO [01-27|10:34:06.028] Reading genesis file          filePath=.tests/precompile/genesis/hello_world.json
wd=/Users/avalabs/go/src/github.com/ava-labs/subnet-evm
INFO [01-27|10:34:06.029] Creating new subnet
[streaming output] [01-27|10:34:06.059] INFO <P Chain> proposervm/pre_fork_block.go:223 built block {"blkID": "vByWDipDCUSwfKogQy2SBzHpfAdd3wsaLyze27NkZ7HuefVbs", "innerBlkID": "2HQHayX2WwvNDoTQtVhTf4ZpaHs1Sm448wUh4DZMSDwfH7smKR", "height": 1, "parentTimestamp": "[08-15|00:00:00.000]", "blockTimestamp": "[01-27|10:34:06.000]"}
INFO [01-27|10:34:06.158] Creating new Subnet-EVM blockchain      genesis="&(Config:(ChainID: 99999 Homestead: 0 EIP150: 0

```

```
ExampleHelloWorld
Contract deployed to: 0x52C84043CD9c865236f11d9Fc9F56aa003c1f922
  ✓ should getHello properly
  ✓ contract should not be able to set greeting without enabled
  ✓ should be add contract to enabled list (4071ms)
    ✓ add contract to enabled list (4071ms)
```

```
• [11.480 seconds]
-----
```

Finally, you will see the load test being skipped as well:

```
Running Suite: subnet-evm small load simulator test suite - /Users/avalabs/go/src/github.com/ava-labs/subnet-evm
=====
Random Seed: 1674833658

Will run 0 of 1 specs
S [SKIPPED]
[Load Simulator]
/Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/load/load_test.go:49
  basic subnet load test [load]
  /Users/avalabs/go/src/github.com/ava-labs/subnet-evm/tests/load/load_test.go:50
-----

Ran 0 of 1 Specs in 0.000 seconds
SUCCESS! -- 0 Passed | 0 Failed | 0 Pending | 1 Skipped
PASS
```

Looks like the tests are passing!

:::note

If your tests failed, please retrace your steps. Most likely the error is that the precompile was not enabled and some code is missing. Try running `npm install` in the contracts directory to ensure that hardhat and other packages are installed.

You may also use the [official tutorial implementation](#) to double-check your work as well.

:::

Running a Local Network

We made it! Everything works in our Ginkgo tests, and now we want to spin up a local network with the Hello World precompile activated.

Start the server in a terminal in a new tab using avalanche-network-runner. Please check out [this link](#) for more information on Avalanche Network Runner, how to download it, and how to use it. The server will be in "listening" mode waiting for API calls.

We will start the server from the Subnet-EVM directory so that we can use a relative file path to the genesis JSON file:

```
cd ${GOPATH}/src/github.com/ava-labs/subnet-evm
avalanche-network-runner server \
--log-level debug \
--port=:8080 \
--grpc-gateway-port=:8081
```

Since we already compiled AvalancheGo and Subnet-EVM in a previous step, we should have the AvalancheGo and Subnet-EVM binaries ready to go.

We can now set the following paths. `AVALANCHEGO_EXEC_PATH` points to the latest AvalancheGo binary we have just built. `AVALANCHEGO_PLUGIN_PATH` points to the plugins path which should have the Subnet-EVM binary we have just built:

```
export AVALANCHEGO_EXEC_PATH="${GOPATH}/src/github.com/ava-labs/avalanchego/build/avalanchego"
export AVALANCHEGO_PLUGIN_PATH="${GOPATH}/src/github.com/ava-labs/avalanchego/build/plugins"
```

The following command will "issue requests" to the server we just spun up. We can use avalanche-network-runner to spin up some nodes that run the latest version of Subnet-EVM:

```
avalanche-network-runner control start \
--log-level debug \
--endpoint="0.0.0.0:8080" \
--number-of-nodes=5 \
--avalanchego-path ${AVALANCHEGO_EXEC_PATH} \
--plugin-dir ${AVALANCHEGO_PLUGIN_PATH} \
--blockchain-specs '[{"vm_name": "subnetevm", "genesis": "./tests/precompile/genesis/hello_world.json"}]
```

We can look at the server terminal tab and see it booting up the local network. If the network startup is successful then you should see something like this:

```
[blockchain RPC for "srEXiWaHuhNyGwPUI444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy"]
"http://127.0.0.1:9650/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU"
[blockchain RPC for "srEXiWaHuhNyGwPUI444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy"]
"http://127.0.0.1:9652/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU"
[blockchain RPC for "srEXiWaHuhNyGwPUI444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy"]
"http://127.0.0.1:9654/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU"
[blockchain RPC for "srEXiWaHuhNyGwPUI444Tu47ZEDwxTWrBQiuD7FmgSAQ6X7Dy"]
"http://127.0.0.1:9656/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU"
```

```
[blockchain RPC for "srEXiWaHuhNyGwPUi444Tu472EDwxTwrbQiuD7FmgSAQ6X7Dy"]
"http://127.0.0.1:9658/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU"
```

This shows the extension to the API server on AvalancheGo that's specific to the Subnet-EVM Blockchain instance. To interact with it, you will want to append the `/rpc` extension, which will supply the standard Ethereum API calls. For example, you can use the RPC URL:

```
http://127.0.0.1:9650/ext/bc/2jDWMrF9yKK8gZfJaaaSfACKeMasiNgHmuZip5mWxUfhKaYoEU/rpc
```

to connect to the blockchain through MetaMask, HardHat, etc.

Conclusion

We have now created a stateful precompile from scratch with the precompile generation tool. We hope you had fun and learned a little more about the Subnet-EVM. Now that you have created a simple stateful precompile, we urge you to create one of your own. If you have an idea for a stateful precompile that may be useful to the community, feel free to create a fork of [Subnet-EVM](#) and create a pull request.

How to Delete a Subnet

Deleting a Subnet Configuration

To delete a created Subnet configuration, run

```
avalanche subnet delete <subnetName>
```

Deleting a Deployed Subnet

You can't delete Subnets deployed to Mainnet or the Fuji Testnet.

However, you may delete Subnets deployed to a local network by cleaning the network state with

```
avalanche network clean
```

How to Import a Subnet into Avalanche-CLI

Context

The most probable reason why someone would want to do this, is if they already deployed a Subnet with Subnet-CLI either to Fuji or Mainnet, and now would like to use Avalanche-CLI to manage the Subnet.

Refer to the [installation guide](#) for instruction about installing Avalanche-CLI.

Because Subnet-CLI is now deprecated, you can use this tutorial to migrate to Avalanche-CLI.

Similarly, you might have created a Subnet "manually" by issuing transactions to node APIs, either a local node or public API nodes, with no help of any tool so far, but would now target Avalanche-CLI integration.

For this How-To, you import the WAGMI Subnet from Fuji.

Requirements

For the import to work properly, you need:

- The Subnet's genesis file, stored on disk
- The Subnet's SubnetID

Import the Subnet

For these use cases, Avalanche-CLI now supports the `import public` command.

Start the import by issuing

```
avalanche subnet import public
```

The tool prompts for the network from which to import. The invariant assumption here is that the network is a public network, either the Fuji testnet or Mainnet. In other words, importing from a local network isn't supported.

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to import from:
  ▶ Fuji
  Mainnet
```

As stated earlier, this is from Fuji, so select it. As a next step, Avalanche-CLI asks for the path of the genesis file on disk:

```
x Provide the path to the genesis file: /tmp/subnet_evm.genesis.json
```

The wizard checks if the file at the provided path exists, refer to the checkmark at the beginning of the line:

```
✓ Provide the path to the genesis file: /tmp/subnetevm_genesis.json
```

Subsequently, the wizard asks if nodes have already been deployed for this Subnet.

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Have nodes already been deployed to this subnet?:
  Yes
▶ No
```

Nodes are Already Validating This Subnet

If nodes already have been deployed, the wizard attempts to query such a node for detailed data like the VM version. This allows the tool to skip querying GitHub (or wherever the VM's repository is hosted) for the VM's version, but rather we'll get the exact version which is actually running on the node.

For this to work, a node API URL is requested from the user, which is used for the query. This requires that the node's API IP and port are accessible from the machine running Avalanche-CLI, or the node is obviously not reachable, and thus the query times out and fails, and the tool exits. The node should also be validating the given Subnet for the import to be meaningful, otherwise, the import fails with missing information.

If the query succeeded, the wizard jumps to prompt for the Subnet ID.

```
Please provide an API URL of such a node so we can query its VM version (e.g. http://111.22.33.44:5555):
http://154.42.240.119:9650
What is the ID of the subnet?: 28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY
```

The rest of the wizard is identical to the next section, except that there is no prompt for the VM version anymore.

Nodes Aren't Yet Validating this Subnet, the Nodes API URL are Unknown, or Inaccessible (Firewalls)

If you don't have a node's API URL at hand, or it's not reachable from the machine running Avalanche-CLI, or maybe no nodes have even been deployed yet because only the `CreateSubnet` transaction has been issued, for example, you can query the public APIs.

You can't know for sure what Subnet VM versions the validators are running though, therefore the tool has to prompt later. So, select `No` when the tool asks for deployed nodes:

Thus, at this point the wizard requests the Subnet's ID, without which it can't know what to import. Remember the ID is different on different networks.

From the [Testnet Subnet Explorer](#) you can see that WAGMI's Subnet ID is `28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY`:

```
✓ What is the ID of the subnet?: 28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyqe79MCv2Mhs8kxiY
```

Notice the checkmark at line start, it signals that there is ID format validation.

If you hit `enter` now, the tool queries the public APIs for the given network, and if successful, it prints some information about the Subnet, and proceeds to ask about the Subnet's type:

```
Getting information from the Fuji network...
Retrieved information. BlockchainID: 2ebCneCbwthjQ1rYT41nhd7M76Hc61mosMAQrTFhBq8qeqh6tt, Name: WAGMI, VMID:
srExiWaHuhNyGwPUi444Tu47ZEDwxTWrbQiuD7FmgSAQ6X7Dy
Use the arrow keys to navigate: ↑ ↓ ← →
? What's this VM's type?:
  ▶ Subnet-EVM
  Custom
```

Avalanche-CLI needs to know the VM type, to hit its repository and select what VM versions are available. This works automatically for Ava Labs VMs (like Subnet-EVM).

Custom VMs aren't supported yet at this point, but are next on the agenda.

As the import is for WAGMI, and you know that it's a Subnet-EVM type, select that.

The tool then queries the (GitHub) repository for available releases, and prompts the user to pick the version she wants to use:

```
✓ Subnet-EVM
Use the arrow keys to navigate: ↑ ↓ ← →
? Pick the version for this VM:
  ▶ v0.4.5
  v0.4.5-rc.1
  v0.4.4
  v0.4.4-rc.0
↓  v0.4.3
```

There is only so much the tool can help here, the Subnet manager/administrator should know what they want to use Avalanche-CLI for, how, and why they're importing the Subnet.

It's crucial to understand that the correct versions are only known to the user. The latest might be usually fine, but the tool can't make assumptions about it easily. This is why it's indispensable that the wizard prompts the user, and the tool requires her to choose.

If you selected to query an actual Subnet validator, not the public APIs, in the preceding step. In such a scenario, the tool skips this picking.

✓ v0.4.5
Subnet WAGMI imported successfully

The choice finalizes the wizard, which hopefully signals that the import succeeded. If something went wrong, the error messages provide cause information. This means you can now use Avalanche-CLI to handle the imported Subnet in the accustomed way. For example, you could deploy the WAGMI Subnet locally.

For a complete description of options, flags, and the command, visit the [command reference](#).

How to View Your Created Subnets

List Subnet Configurations

You can list the Subnets you've created with

avalanche subnet list

Example:

> avalanche subnet list					
SUBNET	CHAIN	CHAIN ID	VM ID	TYPE	FROM REPO
test	test	5234	tGBrM2SXkAdNsqzb3SaFZWMNdzzjFEUKteheTa4dhUwnfQyu	Subnet-EVM	false

To see detailed information about your deployed Subnets, add the `--deployed` flag:

avalanche subnet list --deployed					FUJI
SUBNET (TESTNET)	CHAIN	VM ID	LOCAL NETWORK		
		MAINNET			
test	test	tGBrM2SXkAdNsqzb3SaFZZWMNdzjjFEUKteheTa4dhUwnfQyu	Yes	SubnetID:	
XTK7AM2Pw5A4cCtQ3rTugqbeLCU9mVixML3YwwLYUJ4WXN2Kt		No			
+	+	+		+	+
-----+-----+					
				BlockchainID:	
5ce2WhnyeMELzg9UtfpCDGNwRa2AzMzRhBWFtqmFuiXPWE4TR	No				
-----+-----+					

Describe Subnet Configurations

To see the details of a specific configuration, run

```
avalanche subnet describe <subnetName>
```

Example:

No precompiles set

Viewing a Genesis File

If you'd like to see the raw genesis file, supply the `--genesis` flag.

2010

```
> avalanche subnet describe firstsubnet --genesis
{
  "config": {
    "chainId": 12345,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip150Hash": "0x2086799aaebeae135c246c65021c82b4e15a2c451340993aacfd2751886514f0",
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
```

How to Pause and Resume Local Subnets

If you've deployed a Subnet locally, you can preserve and restore the state of your deployed Subnets.

Stopping the Local Network

To gracefully stop a running local network while preserving state, run

avalanche network stop

When restarted, all of your deployed Subnets resume where they left off.

```
> avalanche network stop  
Network stopped successfully.
```

```
> avalanche network start
Starting previously deployed and stopped snapshot
Booting Network. Wait until healthy...
.....
```

```
Network ready to use. Local network node endpoints:
+-----+-----+
| NODE | VM | URL |
+-----+-----+
| node5 | mySubnet | http://127.0.0.1:9658/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node1 | mySubnet | http://127.0.0.1:9650/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node2 | mySubnet | http://127.0.0.1:9652/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node3 | mySubnet | http://127.0.0.1:9654/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
| node4 | mySubnet | http://127.0.0.1:9656/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc |
+-----+-----+
```

Checking Network Status

If you'd like to determine whether or not a local Avalanche network is running on your machine, run

```
avalanche network status
```

Example Call

If the local network is running, the command prints something like

```
Requesting network status...
Network is Up. Network information:
=====
Healthy: true
Custom VMs healthy: true
Number of nodes: 5
Number of custom VMs: 1
=====
Node information =====
node5 has ID NodeID-P7oB2McjBGgW2NXXWVYjV8JEDFoW9xDE5 and endpoint http://127.0.0.1:9658:
node1 has ID NodeID-7Xhw2mDxuDS44j42TCB6U5579esbst3lg and endpoint http://127.0.0.1:9650:
node2 has ID NodeID-MFrZPVCXPv5iCn6M9K6XduxGTYp891xXZ and endpoint http://127.0.0.1:9652:
node3 has ID NodeID-NFBbbJ4qCmNaCzeW7sxErhvWqvEQMnYcN and endpoint http://127.0.0.1:9654:
node4 has ID NodeID-GWPcbFJZFFzreETSojPlmr846mXEKCt and endpoint http://127.0.0.1:9656:
=====
Custom VM information =====
Endpoint at node4 for blockchain "SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz":
http://127.0.0.1:9656/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Endpoint at node5 for blockchain "SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz":
http://127.0.0.1:9658/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Endpoint at node1 for blockchain "SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz":
http://127.0.0.1:9650/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Endpoint at node2 for blockchain "SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz":
http://127.0.0.1:9652/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
Endpoint at node3 for blockchain "SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz":
http://127.0.0.1:9654/ext/bc/SPqou41AALqxDquEycNYuTJmRvZYbfoV9DYApDJVXKXuwVFPz/rpc
```

If the network isn't running, the command instead prints

```
Requesting network status...
No local network running
```

or

```
Requesting network status...
Error: timed out trying to contact backend controller, it is most probably not running
```

How to Run Avalanche-CLI with Docker

To run Avalanche-CLI in a docker container, you need to enable ipv6.

Edit `/etc/docker/daemon.json`. Add this snippet then restart the docker service.

```
{
  "ipv6": true,
  "fixed-cidr-v6": "fd00::/80"
}
```

How to Transform a Subnet into an Elastic Subnet

Elastic Subnets are permissionless Subnets. More information can be found [here](#).

This how-to guide focuses on taking an already created permissioned Subnet and transforming it to a elastic (or permissionless) Subnet.

Prerequisites

- [Avalanche-CLI installed](#)
- You have deployed a permissioned Subnet on [local](#), on [Fuji](#) or on [Mainnet](#)

Getting Started

In the following commands, make sure to substitute the name of your Subnet configuration for `<subnetName>`.

To transform your permissioned Subnet into an Elastic Subnet (NOTE: this action is irreversible), run

```
avalanche subnet elastic <subnetName>
```

and select the network that you want to transform the Subnet on. Alternatively, you can bypass this prompt by providing the `--local`, `--fuji`, or `--mainnet` flag.

Provide the name and the symbol for the permissionless Subnet's native token. You can also bypass this prompt by providing the `--tokenName` and `--tokenSymbol` flags.

Next, select the Elastic Subnet config. You can choose to use the default values detailed [here](#) or customize the Elastic Subnet config. To bypass the prompt, you can use `--default` flag to use the default Elastic Subnet config.

The command may take a couple minutes to run.

Elastic Subnet Transformation on Fuji and Mainnet

Elastic Subnet transformation on public network requires private key loaded into the tool, or a connected ledger device.

Both stored key usage and ledger usage are enabled on Fuji (see more on creating keys [here](#)) while only ledger usage is enabled on Mainnet (see more on setting up your ledger [here](#)).

To transform a permissioned Subnet into Elastic Subnet on public networks, users are required to provide the keys that control the Subnet defined during the Subnet deployment process (more info on keys in Fuji can be found [here](#), while more info on ledger signing in Mainnet can be found [here](#)).

Results

If all works as expected, you then have the option to automatically transform all existing permissioned validators to permissionless validators.

You can also to skip automatic transformation at this point and choose to manually add permissionless validators later.

You can use the output details such as the Asset ID and Elastic Subnet ID to connect to and interact with your Elastic Subnet.

Adding Permissionless Validators to Elastic Subnet

If you are running this command on local network, you will need to first remove permissioned validators (by running `avalanche subnet removeValidator <subnetName>`) so that you can have a list of local nodes to choose from to be added as a permissionless validator in the Elastic Subnet.

To add permissionless validators to an Elastic Subnet, run

```
avalanche subnet join <subnetName> --elastic
```

You will be prompted with which node you would like to add as a permissionless validator. You can skip this prompt by using `--nodeID` flag.

You will then be prompted with the amount of the Subnet native token that you like to stake in the validator. Alternatively, you can bypass this prompt by providing the `--stake-amount` flag. Note that choosing to add the maximum validator stake amount (defined during Elastic Subnet transformation step above) means that you effectively disable delegation in your validator.

Next, select when the validator will start validating and how long it will be validating for. You can also bypass these prompts by using `--start-time` and `--staking-period` flags.

Adding Permissionless Delegator to a Permissionless Validator in Elastic Subnet

To add permissionless delegators, run

```
avalanche subnet addPermissionlessDelegator <subnetName>
```

You will be prompted with which Subnet validator you would like to delegate to. You can skip this prompt by using `--nodeID` flag.

You will then be prompted with the amount of the Subnet native token that you like to stake in the validator. Alternatively, you can bypass this prompt by providing the `--stake-amount` flag. The amount that can be delegated to a validator is detailed [here](#).

Next, select when you want to start delegating and how long you want to delegate for. You can also bypass these prompts by using `--start-time` and `--staking-period` flags.

How to Upgrade Your Subnet-EVM Precompile Configuration

Upgrading a Subnet

You can customize Subnet-EVM based Subnets after deployment by enabling and disabling precompiles. To do this, create a `upgrade.json` file and place it in the appropriate directory.

This document describes how to perform such network upgrades. It's specific for Subnet-EVM upgrades.

The document [Upgrade a Subnet](#) describes all the background information required regarding Subnet upgrades.

:::warning

It's very important that you have read and understood the previously linked document. Failing to do so can potentially grind your network to a halt.

:::

This tutorial assumes that you have already [installed](#) Avalanche-CLI. It assumes you have already created and deployed a Subnet called `testSubnet`.

Generate the Upgrade File

The [Precompiles](#) documentation describes what files the network upgrade requires, and where to place them.

To generate a valid `upgrade.json` file, run:

```
avalanche subnet upgrade generate testSubnet
```

If you didn't create `testSubnet` yet, you would see this result:

```
avalanche subnet upgrade generate testSubnet
The provided subnet name "testSubnet" does not exist
```

Again, it makes no sense to try the upgrade command if the Subnet doesn't exist. If that's the case, please go ahead and [create](#) the Subnet first.

If the Subnet definition exists, the tool launches a wizard. It may feel a bit redundant, but you first see some warnings, to draw focus to the dangers involved:

```
avalanche subnet upgrade generate testSubnet
Performing a network upgrade requires coordinating the upgrade network-wide.
A network upgrade changes the rule set used to process and verify blocks, such that any node that
upgrades incorrectly or fails to upgrade by the time that upgrade goes into effect may become
out of sync with the rest of the network.

Any mistakes in configuring network upgrades or coordinating them on validators may cause the
network to halt and recovering may be difficult.
Please consult
https://docs.avax.network/subnets/customize-a-subnet#network-upgrades-enabledisable-precompiles
for more information
Use the arrow keys to navigate: ↑ ↓ ← →
? Press [Enter] to continue, or abort by choosing 'no':
▶ Yes
No
```

Go ahead and select `Yes` if you understand everything and you agree.

You see a last note, before the actual configuration wizard starts:

```
Avalanchego and this tool support configuring multiple precompiles.
However, we suggest to only configure one per upgrade.

Use the arrow keys to navigate: ↑ ↓ ← →
? Select the precompile to configure:
▶ Contract Deployment Allow List
  Manage Fee Settings
  Native Minting
  Transaction Allow List
```

Refer to [Precompiles](#) for a description of available precompiles and how to configure them.

Make sure you understand precompiles thoroughly and how to configure them before attempting to continue.

For every precompile in the list, the wizard guides you to provide correct information by prompting relevant questions. For the sake of this tutorial, select `Transaction Allow List`. The document [Restricting Who Can Submit Transactions](#) describes what this precompile is about.

```
✓ Transaction Allow List
Set parameters for the "Manage Fee Settings" precompile
Use the arrow keys to navigate: ↑ ↓ ← →
? When should the precompile be activated?:
  ▶ In 5 minutes
  In 1 day
  In 1 week
  In 2 weeks
  Custom
```

This is actually common to all precompiles: they require an activation timestamp. If you think about it, it makes sense: you want a synchronized activation of your precompile. So think for a moment about when you want to set the activation timestamp to. You can select one of the suggested times in the future, or you can pick a custom one. After picking `Custom`, it shows the following prompt:

```
✓ Custom
✗ Enter the block activation UTC datetime in 'YYYY-MM-DD HH:MM:SS' format:
```

The format is `YYYY-MM-DD HH:MM:SS`, therefore `2023-03-31 14:00:00` would be a valid timestamp. Notice that the timestamp is in UTC. Please make sure you have converted the time from your timezone to UTC. Also notice the `*` at the beginning of the line. The CLI tool does input validation, so if you provide a valid timestamp, the `*` disappears:

```
✗ Enter the block activation UTC datetime in 'YYYY-MM-DD HH:MM:SS' format: 2023-03-31 14:00:00
```

The timestamp must be in the **future**, so make sure you use such a timestamp should you be running this tutorial after `2023-03-31 14:00:00 ;)`.

After you provided the valid timestamp, proceed with the precompile specific configurations:

```
The chosen block activation time is 2023-03-31 14:00:00
Use the arrow keys to navigate: ↑ ↓ ← →
? Add 'adminAddresses'?: 
  ▶ Yes
  No
```

This will enable the addresses added in this section to add other admins and/or add enabled addresses for transaction issuance. The addresses provided in this tutorial are fake.

:::caution

However, make sure you or someone you trust have full control over the addresses. Otherwise, you might bring your Subnet to a halt.

:::

```
✓ Yes
Use the arrow keys to navigate: ↑ ↓ ← →
? Provide 'adminAddresses':
  ▶ Add
  Delete
  Preview
  More Info
  ↓ Done
```

The prompting runs with a pattern used throughout the tool:

- Select an operation:
 - `Add` : adds a new address to the current list
 - `Delete` : removes an address from the current list
 - `Preview` : prints the current list
- `More info` prints additional information for better guidance, if available
- Select `Done` when you completed the list

Go ahead and add your first address:

```
✓ Add
✓ Add an address: 0xaaaabbcccccdddeeefffff1111222233334444
```

Add another one:

```
✓ Add
Add an address: 0xaaaabbcccccdddeeefffff1111222233334444
✓ Add
✓ Add an address: 0x1111222233334444aaaabbcccccdddeeefffff
```

Select `Preview` this time to confirm the list is correct:

```
✓ Preview
0. 0xaaaabbCCccCDDdEeEFFFFf1111222233334444
1. 0x1111222233334444aAaAbBbCCCDdDDeEeEfffff
Use the arrow keys to navigate: ↑ ↓ ← →
? Provide 'adminAddresses':
  ▶ Add
  Delete
  Preview
  More Info
  ↓ Done
```

If it looks good, select `Done` to continue:

```
✓ Done
Use the arrow keys to navigate: ↑ ↓ ← →
? Add 'enabledAddresses'?:  
▶ Yes  
No
```

Add one such enabled address, these are addresses which can issue transactions:

```
✓ Add
✓ Add an address: 0x55554444333222111eeeeaaaabbcccccddd█
```

After you added this address, and selected `Done`, the tool asks if you want to add another precompile:

```
✓ Done
Use the arrow keys to navigate: ↑ ↓ ← →
? Should we configure another precompile?:  
▶ No  
Yes
```

If you needed to add another one, you would select `Yes` here. The wizard would guide you through the other available precompiles, excluding already configured ones.

To avoid making this tutorial too long, the assumption is you're done here. Select `No`, which ends the wizard.

This means you have successfully terminated the generation of the upgrade file, often called upgrade bytes. The tool stores them internally.

:::warning

You shouldn't move files around manually. Use the `export` and `import` commands to get access to the files.

:::

So at this point you can either:

- Deploy your upgrade bytes locally
- Export your upgrade bytes to a file, for installation on a validator running on another machine
- Import a file into a different machine running Avalanche-CLI

How To Upgrade a Local Network

The normal use case for this operation is that:

- You already created a Subnet
- You already deployed the Subnet locally
- You already generated the upgrade file with the preceding command or imported into the tool
- This tool already started the network

If the preceding requirements aren't met, the network upgrade command fails.

Therefore, to apply your generated or imported upgrade configuration:

```
avalanche subnet upgrade apply testSubnet
```

A number of checks run. For example, if you created the Subnet but didn't deploy it locally:

```
avalanche subnet upgrade apply testSubnet
Error: no deployment target available
Usage:
  avalanche subnet upgrade apply [subnetName] [flags]

Flags:
  --avalanchego-chain-config-dir string  avalanchego's chain config file directory (default
  "/home/fabio/.avalanchego/chains")
    --config                                create upgrade config for future subnet deployments (same as generate)
    --fuji fuji                             apply upgrade existing fuji deployment (alias for `testnet`)
  -h, --help                               help for apply
    --local local                           apply upgrade existing local deployment
    --mainnet mainnet                      apply upgrade existing mainnet deployment
    --print                                 if true, print the manual config without prompting (for public networks only)
    --testnet testnet                        apply upgrade existing testnet deployment (alias for `fuji`)

Global Flags:
  --log-level string  log level for the application (default "ERROR")
```

Go ahead and [deploy](#) first your Subnet if that's your case.

If you already had deployed the Subnet instead, you see something like this:

```

avalanche subnet upgrade apply testSubnet
Use the arrow keys to navigate: ↑ ↓ ← →
? What deployment would you like to upgrade:
  ▶ Existing local deployment

```

Select `Existing local deployment`. This installs the upgrade file on all nodes of your local network running in the background.

Et voilà. This is the output shown if all went well:

```

✓ Existing local deployment
.....
Network restarted and ready to use. Upgrade bytes have been applied to running nodes at these endpoints.
The next upgrade will go into effect 2023-03-31 09:00:00
+-----+
| NODE | VM | URL |
+-----+
| node1 | testSubnet | http://0.0.0.0:9650/ext/bc/2YTRV2roEhgvwJz7D7vr33hUZscpaZgcYgUTjeMK9KH99NFnsH/rpc |
+-----+
| node2 | testSubnet | http://0.0.0.0:9652/ext/bc/2YTRV2roEhgvwJz7D7vr33hUZscpaZgcYgUTjeMK9KH99NFnsH/rpc |
+-----+
| node3 | testSubnet | http://0.0.0.0:9654/ext/bc/2YTRV2roEhgvwJz7D7vr33hUZscpaZgcYgUTjeMK9KH99NFnsH/rpc |
+-----+
| node4 | testSubnet | http://0.0.0.0:9656/ext/bc/2YTRV2roEhgvwJz7D7vr33hUZscpaZgcYgUTjeMK9KH99NFnsH/rpc |
+-----+
| node5 | testSubnet | http://0.0.0.0:9658/ext/bc/2YTRV2roEhgvwJz7D7vr33hUZscpaZgcYgUTjeMK9KH99NFnsH/rpc |
+-----+

```

There is only so much the tool can do here for you. It installed the upgrade bytes *as-is* as you configured respectively provided them to the tool. You should verify yourself that the upgrades were actually installed correctly, for example issuing some transactions - mind the timestamp!

Apply the Upgrade to a Public Node (Fuji or Mainnet)

For this scenario to work, you should also have deployed the Subnet to the public network (Fuji or Mainnet) with this tool. Otherwise, the tool won't know the details of the Subnet, and won't be able to guide you.

Assuming the Subnet has been already deployed to Fuji, when running the `apply` command, the tool notices the deployment:

```

avalanche subnet upgrade apply testSubnet
Use the arrow keys to navigate: ↑ ↓ ← →
? What deployment would you like to upgrade:
  Existing local deployment
  ▶ Fuji

```

If not, you would not find the `Fuji` entry here.

:::important

This scenario assumes that you are running the `fuji` validator on the same machine which is running Avalanche-CLI.

:::

If this is the case, the tool tries to install the upgrade file at the expected destination. If you use default paths, it tries to install at `$HOME/.avalanchego/chains/`, creating the chain id directory, so that the file finally ends up at `$HOME/.avalanchego/chains/<chain-id>/upgrade.json`.

If you are *not* using default paths, you can configure the path by providing the flag `--avalanchego-chain-config-dir` to the tool. For example:

```
avalanche subnet upgrade apply testSubnet --avalanchego-chain-config-dir /path/to/your/chains
```

Make sure to identify correctly where your chain config dir is, or the node might fail to find it.

If all is correct, the file gets installed:

```

avalanche subnet upgrade apply testSubnet
✓ Fuji
The chain config dir avalanchego uses is set at /home/fabio/.avalanchego/chains
Trying to install the upgrade files at the provided /home/fabio/.avalanchego/chains path
Successfully installed upgrade file

```

If however the node is *not* running on this same machine where you are executing Avalanche-CLI, there is no point in running this command for a Fuji node. In this case, you might rather export the file and install it at the right location.

To see the instructions about how to go about this, add the `--print` flag:

```

avalanche subnet upgrade apply testSubnet --print
✓ Fuji
To install the upgrade file on your validator:

```

```

1. Identify where your validator has the avalanchego chain config dir configured.
The default is at $HOME/.avalanchego/chains (/home/user/.avalanchego/chains on this machine).
If you are using a different chain config dir for your node, use that one.
2. Create a directory with the blockchainID in the configured chain-config-dir (e.g.
$HOME/.avalanchego/chains/ExDKhjXq1Vg7s35p8YJ56CJpcw6nJgcGCCE7DbQ4oBKnZlqxi) if doesn't already exist.
3. Create an `upgrade.json` file in the blockchain directory with the content of your upgrade file.
This is the content of your upgrade file as configured in this tool:
{
  "precompileUpgrades": [
    {
      "txAllowListConfig": {
        "adminAddresses": [
          "0xb3d82b1367d362de99ab59a658165aff520cbd4d"
        ],
        "enabledAddresses": null,
        "blockTimestamp": 1677550447
      }
    }
  ]
}

*****
* Upgrades are tricky. The syntactic correctness of the upgrade file is important. *
* The sequence of upgrades must be strictly observed. *
* Make sure you understand https://docs.avax.network/nodes/maintain/chain-config-flags#subnet-chain-configs *
* before applying upgrades manually. *
*****

```

The instructions also show the content of your current upgrade file, so you can just select that if you wish. Or actually export the file.

Export the Upgrade File

If you have generated the upgrade file, you can export it:

```

avalanche subnet upgrade export testSubnet
✓ Provide a path where we should export the file to: /tmp/testSubnet-upgrade.json

```

Just provide a valid path to the prompt, and the tool exports the file there.

```

avalanche subnet upgrade export testSubnet
Provide a path where we should export the file to: /tmp/testSubnet-upgrade.json
Writing the upgrade bytes file to "/tmp/testSubnet-upgrade.json"...
File written successfully.

```

You can now take that file and copy it to validator nodes, see preceding instructions.

Import the Upgrade File

You or someone else might have generated the file elsewhere, or on another machine. And now you want to install it on the validator machine, using Avalanche-CLI.

You can import the file:

```

avalanche subnet upgrade import testSubnet
Provide the path to the upgrade file to import: /tmp/testSubnet-upgrade.json

```

An existing file with the same path and filename would be overwritten.

After you have imported the file, you can `apply` it either to a local network or to a locally running validator. Follow the instructions for the appropriate use case.

How to Upgrade a Subnet's Virtual Machine

This how-to guide explains how to upgrade a locally deployed Subnet. Upgrading Fuji and Mainnet Subnets is currently not supported.

Upgrading a Local VM

To upgrade a local Subnet, you first need to pause the local network. To do so, run

```
avalanche network stop
```

Next, you need to select the new VM to run your Subnet on. If you're running a Subnet-EVM Subnet, you likely want to bump to the latest released version. If you're running a Custom VM, you'll want to choose another custom binary.

Start the upgrade wizard with

```
avalanche subnet upgrade vm <subnetName>
```

where you replace `<subnetName>` with the name of the Subnet you'd like to upgrade.

Selecting a VM Deployment to Upgrade

After starting the Subnet Upgrade Wizard, you should see something like this:

```
? What deployment would you like to upgrade:  
▶ Update config for future deployments  
Existing local deployment
```

If you select the first option, Avalanche-CLI updates your Subnet's config and any future calls to `avalanche subnet deploy` use the new version you select. However, any existing local deployments continue to use the old version.

If you select the second option, the opposite occurs. The existing local deployment switches to the new VM but subsequent deploys use the original.

Select a VM to Upgrade To

The next option asks you to select your new virtual machine.

```
? How would you like to update your subnet's virtual machine:  
▶ Update to latest version  
Update to a specific version  
Update to a custom binary
```

If you're using the Subnet-EVM, you'll have the option to upgrade to the latest released version. You can also select a specific version or supply a custom binary. If your Subnet already uses a custom VM, you need to select another custom binary.

Once you select your VM, you should see something like:

```
Upgrade complete. Ready to restart the network.
```

Restart the Network

:::note

If you are running multiple Subnets concurrently, you may need to update multiple Subnets to restart the network. All of your deployed must be using the same RPC Protocol version. You can see more details about this [here](#).

:::

Finally, restart the network with

```
avalanche network start
```

If the network starts correctly, your Subnet is now running the upgraded VM.

Install Avalanche-CLI

Compatibility

Avalanche-CLI runs on Linux and Mac. Windows is currently not supported.

Instructions

To download a binary for the latest release, run:

```
curl -sSfL https://raw.githubusercontent.com/ava-labs/avalanche-cli/main/scripts/install.sh | sh -s
```

The script installs the binary inside the `~/bin` directory. If the directory doesn't exist, it will be created.

Adding Avalanche-CLI to Your PATH

To call the `avalanche` binary from anywhere, you'll need to add it to your system path. If you installed the binary into the default location, you can run the following snippet to add it to your path.

```
export PATH=~/bin:$PATH
```

To add it to your path permanently, add an export command to your shell initialization script. If you run `bash`, use `.bashrc`. If you run `zsh`, use `.zshrc`.

For example:

```
export PATH=~/bin:$PATH >> .bashrc
```

Checking Your Installation

You can test your installation by running `avalanche --version`. The tool should print the running version.

Updating

To update your installation, you need to delete your current binary and download the latest version using the preceding steps.

Building from Source

The source code is available in this [GitHub repository](#).

After you've cloned the repository, checkout the tag you'd like to run. You can compile the code by running `./scripts/build.sh` from the top level directory.

The build script names the binary `./bin/avalanche`.

Introduction to VMs

Overview

This is part of a series of tutorials for building a Virtual Machine (VM):

- Introduction to VMs (this article)
- [How to Build a Simple Golang VM](#)
- [How to Build a Complex Golang VM](#)
- [How to Build a Simple Rust VM](#)

A [Virtual Machine \(VM\)](#) is a blueprint for a blockchain. Blockchains are instantiated from a VM, similar to how objects are instantiated from a class definition. VMs can define anything you want, but will generally define transactions that are executed and how blocks are created.

Blocks and State

Virtual Machines deal with blocks and state. The functionality provided by VMs is to:

- Define the representation of a blockchain's state
- Represent the operations in that state
- Apply the operations in that state

Each block in the blockchain contains a set of state transitions. Each block is applied in order from the blockchain's initial genesis block to its last accepted block to reach the latest state of the blockchain.

Blockchain

A blockchain relies on two major components: The **Consensus Engine** and the **VM**. The VM defines application specific behavior and how blocks are built and parsed to create the blockchain. VMs all run on top of the Avalanche Consensus Engine, which allows nodes in the network to agree on the state of the blockchain. Here's a quick example of how VMs interact with consensus:

1. A node wants to update the blockchain's state
2. The node's VM will notify the consensus engine that it wants to update the state
3. The consensus engine will request the block from the VM
4. The consensus engine will verify the returned block using the VM's implementation of `Verify()`
5. The consensus engine will get the network to reach consensus on whether to accept or reject the newly verified block
 - Every virtuous (well-behaved) node on the network will have the same preference for a particular block
6. Depending upon the consensus results, the engine will either accept or reject the block
 - What happens when a block is accepted or rejected is specific to the implementation of the VM

AvalancheGo provides the consensus engine for every blockchain on the Avalanche Network. The consensus engine relies on the VM interface to handle building, parsing, and storing blocks as well as verifying and executing on behalf of the consensus engine.

This decoupling between the application and consensus layer allows developers to build their applications quickly by implementing virtual machines, without having to worry about the consensus layer managed by Avalanche which deals with how nodes agree on whether or not to accept a block.

Installing a VM

VMs are supplied as binaries to a node running `AvalancheGo`. These binaries must be named the VM's assigned **VMID**. A VMID is a 32-byte hash encoded in CB58 that is generated when you build your VM.

In order to install a VM, its binary must be installed in the `AvalancheGo` plugin path. See [here](#) for more details. Multiple VMs can be installed in this location.

Each VM runs as a separate process from AvalancheGo and communicates with `AvalancheGo` using gRPC calls. This is functionality is enabled by `rpcchainvm`, a special VM that wraps around other VM implementations so that they can communicate back and forth with the AvalancheGo.

API Handlers

Users can interact with a blockchain and its VM through handlers exposed by the VM's API.

VMs expose two types of handlers to serve responses for incoming requests:

- **Blockchain Handlers** - Referred to as handlers, these expose APIs to interact with a blockchain instantiated by a VM. The API endpoint will be different for each chain. The endpoint for a handler is `/ext/bc/[chainID]` .
- **VM Handlers** - Referred to as static handlers, these expose APIs to interact with the VM directly. One example API would be to parse genesis data to instantiate a new blockchain. The endpoint for a static handler is `/ext/vm/[vmID]` .

For any readers familiar with object-oriented programming, static and non-static handlers on a VM are analogous to static and non-static methods on a class. Blockchain handlers can be thought of as methods on an object, whereas VM handlers can be thought of as static methods on a class.

Instantiate a VM

The `vm.Factory` interface is implemented to create new VM instances from which a blockchain can be initialized. The factory's `New` method shown below provides `AvalancheGo` with an instance of the VM. It's defined in the [factory.go](#) file of the `timestampvm` repository.

```
// Returning a new VM instance from VM's factory
func (f *Factory) New(*snow.Context) (interface{}, error) { return &vm.VM{}, nil }
```

Initializing a VM to Create a Blockchain

Before a VM can run, `AvalancheGo` will initialize it by invoking its `Initialize` method. Here, the VM will bootstrap itself and sets up anything it requires before it starts running.

This might involve setting up its database, mempool, genesis state, or anything else the VM requires to run.

```
if err := vm.Initialize(
    ctx.Context,
    vmDBManager,
    genesisData,
    chainConfig.Upgrade,
    chainConfig.Config,
    msgChan,
    fxs,
    sender,
);
```

You can refer to the [implementation](#) of `vm.initialize` in the `TimestampVM` repository.

Interfaces

Every VM should implement the following interfaces:

`block.ChainVM`

To reach a consensus on linear blockchains, Avalanche uses the Snowman consensus engine. To be compatible with Snowman, a VM must implement the `block.ChainVM` interface.

For more information, see [here](#).

```
// ChainVM defines the required functionality of a Snowman VM.
//
// A Snowman VM is responsible for defining the representation of the state,
// the representation of operations in that state, the application of operations
// on that state, and the creation of the operations. Consensus will decide on
// if the operation is executed and the order operations are executed.
//
// For example, suppose we have a VM that tracks an increasing number that
// is agreed upon by the network.
// The state is a single number.
// The operation is setting the number to a new, larger value.
// Applying the operation will save to the database the new value.
// The VM can attempt to issue a new number, of larger value, at any time.
// Consensus will ensure the network agrees on the number at every block height.
type ChainVM interface {
    common.VM
    Getter
    Parser

    // Attempt to create a new block from data contained in the VM.
    //
    // If the VM doesn't want to issue a new block, an error should be
    // returned.
    BuildBlock() (snowman.Block, error)

    // Notify the VM of the currently preferred block.
    //
    // This should always be a block that has no children known to consensus.
    SetPreference(ids.ID) error

    // LastAccepted returns the ID of the last accepted block.
}
```

```

// If no blocks have been accepted by consensus yet, it is assumed there is
// a definitionally accepted block, the Genesis block, that will be
// returned.
LastAccepted() (ids.ID, error)
}

// Getter defines the functionality for fetching a block by its ID.
type Getter interface {
    // Attempt to load a block.
    //
    // If the block does not exist, an error should be returned.
    //
    GetBlock(ids.ID) (snowman.Block, error)
}

// Parser defines the functionality for fetching a block by its bytes.
type Parser interface {
    // Attempt to create a block from a stream of bytes.
    //
    // The block should be represented by the full byte array, without extra
    // bytes.
    ParseBlock([]byte) (snowman.Block, error)
}

```

common.VM

`common.VM` is a type that every `VM` must implement.

For more information, you can see the full file [here](#).

```

// VM describes the interface that all consensus VMs must implement
type VM interface {
    // Contains handlers for VM-to-VM specific messages
    AppHandler

    // Returns nil if the VM is healthy.
    // Periodically called and reported via the node's Health API.
    health.Checkable

    // Connector represents a handler that is called on connection connect/disconnect
    validators.Connector

    // Initialize this VM.
    // [ctx]: Metadata about this VM.
    //     [ctx.networkID]: The ID of the network this VM's chain is running on.
    //     [ctx.chainID]: The unique ID of the chain this VM is running on.
    //     [ctx.Log]: Used to log messages
    //     [ctx.NodeID]: The unique staker ID of this node.
    //     [ctx.Lock]: A Read/Write lock shared by this VM and the consensus
    //                 engine that manages this VM. The write lock is held
    //                 whenever code in the consensus engine calls the VM.
    //     [dbManager]: The manager of the database this VM will persist data to.
    //     [genesisBytes]: The byte-encoding of the genesis information of this
    //                     VM. The VM uses it to initialize its state. For
    //                     example, if this VM were an account-based payments
    //                     system, `genesisBytes` would probably contain a genesis
    //                     transaction that gives coins to some accounts, and this
    //                     transaction would be in the genesis block.
    //     [toEngine]: The channel used to send messages to the consensus engine.
    //     [fxs]: Feature extensions that attach to this VM.
    Initialize(
        ctx *snow.Context,
        dbManager manager.Manager,
        genesisBytes []byte,
        upgradeBytes []byte,
        configBytes []byte,
        toEngine chan<- Message,
        fxs []*Fx,
        appSender AppSender,
    ) error

    // Bootstrapping is called when the node is starting to bootstrap this chain.
    Bootstrapping() error

    // Bootstrapped is called when the node is done bootstrapping this chain.
    Bootstrapped() error
}

```

```

// Shutdown is called when the node is shutting down.
Shutdown() error

// Version returns the version of the VM this node is running.
Version() (string, error)

// Creates the HTTP handlers for custom VM network calls.
//
// This exposes handlers that the outside world can use to communicate with
// a static reference to the VM. Each handler has the path:
// [Address of node]/ext/VM/[VM ID]/[extension]
//
// Returns a mapping from [extension]s to HTTP handlers.
//
// Each extension can specify how locking is managed for convenience.
//
// For example, it might make sense to have an extension for creating
// genesis bytes this VM can interpret.
CreateStaticHandlers() (map[string]*HTTPHandler, error)

// Creates the HTTP handlers for custom chain network calls.
//
// This exposes handlers that the outside world can use to communicate with
// the chain. Each handler has the path:
// [Address of node]/ext/bc/[chain ID]/[extension]
//
// Returns a mapping from [extension]s to HTTP handlers.
//
// Each extension can specify how locking is managed for convenience.
//
// For example, if this VM implements an account-based payments system,
// it have an extension called `accounts`, where clients could get
// information about their accounts.
CreateHandlers() (map[string]*HTTPHandler, error)
}

```

snowman.Block

The `snowman.Block` interface It define the functionality a block must implement to be a block in a linear Snowman chain.

For more information, you can see the full file [here](#).

```

// Block is a possible decision that dictates the next canonical block.
//
// Blocks are guaranteed to be Verified, Accepted, and Rejected in topological
// order. Specifically, if Verify is called, then the parent has already been
// verified. If Accept is called, then the parent has already been accepted. If
// Reject is called, the parent has already been accepted or rejected.
//
// If the status of the block is Unknown, ID is assumed to be able to be called.
// If the status of the block is Accepted or Rejected; Parent, Verify, Accept,
// and Reject will never be called.
type Block interface {
    choices.Decidable

    // Parent returns the ID of this block's parent.
    Parent() ids.ID

    // Verify that the state transition this block would make if accepted is
    // valid. If the state transition is invalid, a non-nil error should be
    // returned.
    //
    // It is guaranteed that the Parent has been successfully verified.
    Verify() error

    // Bytes returns the binary representation of this block.
    //
    // This is used for sending blocks to peers. The bytes should be able to be
    // parsed into the same block on another node.
    Bytes() []byte

    // Height returns the height of this block in the chain.
    Height() uint64
}

```

choices.Decidable

This interface is a superset of every decidable object, such as transactions, blocks, and vertices.

For more information, you can see the full file [here](#).

```
// Decidable represents element that can be decided.  
//  
// Decidable objects are typically thought of as either transactions, blocks, or  
// vertices.  
type Decidable interface {  
    // ID returns a unique ID for this element.  
    //  
    // Typically, this is implemented by using a cryptographic hash of a  
    // binary representation of this element. An element should return the same  
    // IDs upon repeated calls.  
    ID() ids.ID  
  
    // Accept this element.  
    //  
    // This element will be accepted by every correct node in the network.  
    Accept() error  
  
    // Reject this element.  
    //  
    // This element will not be accepted by any correct node in the network.  
    Reject() error  
  
    // Status returns this element's current status.  
    //  
    // If Accept has been called on an element with this ID, Accepted should be  
    // returned. Similarly, if Reject has been called on an element with this  
    // ID, Rejected should be returned. If the contents of this element are  
    // unknown, then Unknown should be returned. Otherwise, Processing should be  
    // returned.  
    Status() Status  
}
```

How to Deploy a Subnet with Multisig Authorization

Subnet creators can control critical Subnet operations with a N of M multisig. This multisig must be setup at deployment time and can't be edited afterward. Multisigs can be available on both the Fuji Testnet and Mainnet.

To setup your multisig, you need to know the P-Chain address of each key holder and what you want your signing threshold to be.

:::note

Avalanche-CLI requires Ledgers for Mainnet deployments. This how-to guide assumes the use of Ledgers for setting up your multisig.

:::

Prerequisites

- [Avalanche-CLI](#) installed
- Familiarity with process of [Deploying a Subnet on Testnet](#) and [Deploying a Permissioned Subnet on Mainnet](#)
- Multiple Ledger devices [configured for Avalanche](#)
- A Subnet configuration ready to deploy to either Fuji Testnet or Mainnet

Deploy a Subnet with a Multisig

When issuing the transactions to create the Subnet, you need to sign the TXs with multiple keys from the multisig.

Specify Network

Start the Subnet deployment with

```
avalanche subnet deploy testsubnet
```

First step is to specify Fuji or Mainnet as the network:

```
Use the arrow keys to navigate: ↑ ↓ ← →  
? Choose a network to deploy on:  
  Local Network  
  Fuji  
  ▶ Mainnet
```

```
Deploying [testsubnet] to Mainnet
```

Ledger is automatically recognized as the signature mechanism on `Mainnet`.

After that, the CLI shows the first `Mainnet` Ledger address.

```
Ledger address: P-avax1kdzq569g2c9urm9887cmldlsa3w3jhxe0knfy5
```

Set Control Keys

Next the CLI asks the user to specify the control keys. This is where you setup your multisig.

```
Configure which addresses may make changes to the subnet.  
These addresses are known as your control keys. You are going to also  
set how many control keys are required to make a subnet change (the threshold).  
Use the arrow keys to navigate: ↑ ↓ ← →  
? How would you like to set your control keys?:  
▶ Use ledger address  
Custom list
```

Select `Custom list` and add every address that you'd like to be a key holder on the multisig.

```
✓ Custom list  
? Enter control keys:  
▶ Add  
Delete  
Preview  
More Info  
↓ Done
```

Use the given menu to add each key, and select `Done` when finished.

The output at this point should look something like:

```
✓ Custom list  
✓ Add  
Enter P-Chain address (Example: P-...): P-avax1wryu62weky9qjlp40cpmnqf6ml2hytnagj5q28  
✓ Add  
Enter P-Chain address (Example: P-...): P-avax1kdzq569g2c9urm9887cmldlsa3w3jhxe0knfy5  
✓ Add  
Enter P-Chain address (Example: P-...): P-avax12gcy0x10al6gcjrt0395xqlcuq078m193wl5h8  
✓ Add  
Enter P-Chain address (Example: P-...): P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af  
✓ Add  
Enter P-Chain address (Example: P-...): P-avax1g4eryh40dtscltmxn9zk925ny07gdq2xyjtf4g  
✓ Done  
Your Subnet's control keys: [P-avax1wryu62weky9qjlp40cpmnqf6ml2hytnagj5q28 P-avax1kdzq569g2c9urm9887cmldlsa3w3jhxe0knfy5 P-avax12gcy0x10al6gcjrt0395xqlcuq078m193wl5h8 P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af P-avax1g4eryh40dtscltmxn9zk925ny07gdq2xyjtf4g]
```

:::note

When deploying a Subnet with Ledger's, you must include the Ledger's default address determined in [Specify Network](#) for the deployment to succeed. You may see an error like

```
Error: wallet does not contain subnet auth keys  
exit status 1
```

:::

Set Threshold

Next, specify the threshold. In your N of M multisig, your threshold is N, and M is the number of control keys you added in the previous step.

```
Use the arrow keys to navigate: ↑ ↓ ← →  
? Select required number of control key signatures to make a subnet change:  
▶ 1  
2  
3  
4  
5
```

Specify Control Keys to Sign the Chain Creation TX

You now need N of your key holders to sign the Subnet deployment transaction. You must select which addresses you want to sign the TX.

```
? Choose a subnet auth key:
▶ P-avax1wryu62weky9qjlp40cpmnqf6ml2hytnagj5q28
P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5
P-avax12gcy0x10a16gcjrt0395xqlcuq078ml93w15h8
P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
P-avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g
```

A successful control key selection looks like:

```
✓ 2
✓ P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5
✓ P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
Your subnet auth keys for chain creation: [P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5 P-
avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af]
*** Please sign subnet creation hash on the ledger device ***
```

Potential Errors

If the currently connected Ledger address isn't included in your TX signing group, the operation fails with:

```
✓ 2
✓ P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
✓ P-avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g
Your subnet auth keys for chain creation: [P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af P-
avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g]
Error: wallet does not contain subnet auth keys
exit status 1
```

This can happen either because the original specified control keys -previous step- don't contain the Ledger address, or because the Ledger address control key wasn't selected in the current step.

If the user has the correct address but doesn't have sufficient balance to pay for the TX, the operation fails with:

```
✓ 2
✓ P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
✓ P-avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g
Your subnet auth keys for chain creation: [P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af P-
avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g]
*** Please sign subnet creation hash on the ledger device ***
Error: insufficient funds: provided UTXOs need 1000000000 more units of asset
"rgNLkDPpANwqg3pHC4o9aGJmf2YU4GgTVUMRKAdnKodihkqgr"
exit status 1
```

Sign Subnet Deployment TX with the First Address

The Subnet Deployment TX is ready for signing.

```
*** Please sign subnet creation hash on the ledger device ***
```

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

```
Subnet has been created with ID: 2qUKjvPx68Fgc1NMi8w4mtaBt5hStgBzPhsQrSlm7vSub2q9ew. Now creating blockchain...
*** Please sign blockchain creation hash on the ledger device ***
```

After successful Subnet creation, the CLI asks the user to sign the blockchain creation TX.

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

On success, the CLI provides Subnet deploy details. As only one address signed the chain creation TX, the CLI writes a file to disk to save the TX to continue the signing process with another command.

DEPLOYMENT RESULTS	
Chain Name	testsubnet
Subnet ID	2qUKjvPx68Fgc1NMi8w4mtaBt5hStgBzPhsQrSlm7vSub2q9ew
VM ID	rW1esjm6gy4BtGvxKMpHB2M28MJGFNs9HRY9AmnchhdceB3ii

```
1 of 2 required Blockchain Creation signatures have been signed. Saving TX to disk to enable remaining signing.
```

```
Path to export partially signed TX to:
```

Enter the name of file to write to disk, such as `partiallySigned.txt`. This file shouldn't exist already.

```
Path to export partially signed TX to: partiallySigned.txt
```

Addresses remaining to sign the tx

```
P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
```

Connect a ledger with one of the remaining addresses or choose a stored key and run the signing command, or send "partiallySigned.txt" to another user for signing.

Signing command:

```
avalanche transaction sign testsubnet --input-tx-filepath partiallySigned.txt
```

Gather Remaining Signatures and Issue the Subnet Deployment TX

So far, one address has signed the Subnet deployment TX, but you need N signatures. Your Subnet has not been fully deployed yet. To get the remaining signatures, you may connect a different Ledger to the same computer you've been working on. Alternatively, you may send the `partiallySigned.txt` file to other users to sign themselves.

The remainder of this section assumes that you are working on a machine with access to both the remaining keys and the `partiallySigned.txt` file.

Issue the Command to Sign the Chain Creation TX

Avalanche-CLI can detect the deployment network automatically. For `Mainnet` TXs, it uses your Ledger automatically. For `Fuji Testnet`, the CLI prompts the user to choose the signing mechanism.

You can start the signing process with the `transaction sign` command:

```
avalanche transaction sign testsubnet --input-tx-filepath partiallySigned.txt
```

```
Ledger address: P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
*** Please sign TX hash on the ledger device ***
```

Next, the CLI starts a new signing process for the Subnet deployment TX. If the Ledger isn't the correct one, the following error should appear instead:

```
Ledger address: P-avax1kdzq569g2c9urm9887cmldlsa3w3jhxe0knfy5
Error: wallet does not contain subnet auth keys
exit status 1
```

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

Repeat this processes until all required parties have signed the TX. You should see a message like this:

```
All 2 required Tx signatures have been signed. Saving TX to disk to enable commit.

Overwriting partiallySigned.txt

Tx is fully signed, and ready to be committed

Commit command:
avalanche transaction commit testsubnet --input-tx-filepath partiallySigned.txt
```

Now, `partiallySigned.txt` contains a fully signed TX.

Commit the Subnet Deployment TX

To run submit the fully signed TX, run

```
avalanche transaction commit testsubnet --input-tx-filepath partiallySigned.txt
```

The CLI recognizes the deployment network automatically and submits the TX appropriately.

DEPLOYMENT RESULTS	
Chain Name	testsubnet
Subnet ID	2qUKjvPx68Fgc1NM18w4mtaBt5hStgBzPhsQrs1m7vSub2q9ew
VM ID	rW1esjm6gy4BtGvxKMpHB2M28MJGFNsqHRY9AmnchdcgeB3ii

```
+-----+-----+
| Blockchain ID | 2fx9EF61C964cWBu55vcz9b7gH9LFBkPwoj49JTSHA6Soqqzoj |
+-----+-----+
| RPC URL      | http://127.0.0.1:9650/ext/bc/2fx9EF61C964cWBu55vcz9b7gH9LFBkPwoj49JTSHA6Soqqzoj/rpc |
+-----+-----+
| P-Chain TXID | 2fx9EF61C964cWBu55vcz9b7gH9LFBkPwoj49JTSHA6Soqqzoj |
+-----+-----+
```

Your Subnet successfully deployed with a multisig.

Add Validators Using the Multisig

The `addValidator` command also requires use of the multisig. Before starting, make sure to connect, unlock, and run the Avalanche Ledger app.

```
avalanche subnet addValidator testsubnet
```

Select Network

First specify the network. Select either `Fuji` or `Mainnet`.

```
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a network to add validator to.:
▶ Fuji
Mainnet
```

Choose Signing Keys

Then, similar to the `deploy` command, the command asks the user to select the N control keys needed to sign the TX.

```
✓ Mainnet
Use the arrow keys to navigate: ↑ ↓ ← →
? Choose a subnet auth key:
▶ P-avax1wryu62weky9qjlp40cpmmnf6ml2hytnajj5q28
P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5
P-avax12gcy0x10a16gcjrt0395xqlcuq078m193w15h
P-avax1g7nkguzug8yju8cq3ndzc9lql2yg69s9ejqa2af
P-avax1g4eryh40dtcs1tmxn9zk925ny07gdq2xyjtf4g

✓ Mainnet
✓ P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5
✓ P-avax1g7nkguzug8yju8cq3ndzc9lql2yg69s9ejqa2af
Your subnet auth keys for add validator TX creation: [P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5 P-
avax1g7nkguzug8yju8cq3ndzc9lql2yg69s9ejqa2af].
```

Finish Assembling the TX

Take a look at [Add a Validator](#) for additional help issuing this transaction.

:::note

If setting up a multisig, don't select your validator start time to be in one minute. Finishing the signing process takes significantly longer when using a multisig.

:::

Next, we need the NodeID of the validator you want to whitelist.

```
Check https://docs.avax.network/apis/avalanchego/apis/info#infogetnodeid for instructions about how to query the NodeID from
your node
(Edit host IP address and port to match your deployment, if needed).
What is the NodeID of the validator you'd like to whitelist?: NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg
✓ Default (20)
When should your validator start validating?
If you validator is not ready by this time, subnet downtime can occur.
✓ Custom
When should the validator start validating? Enter a UTC datetime in 'YYYY-MM-DD HH:MM:SS' format: 2022-11-22 23:00:00
✓ Until primary network validator expires
NodeID: NodeID-7Xhw2mDxuDS44j42TCB6U5579esbSt3Lg
Network: Local Network
Start time: 2022-11-22 23:00:00
End time: 2023-11-22 15:57:27
Weight: 20
Inputs complete, issuing transaction to add the provided validator information...
```

Ledger address: P-avax1kdqz569g2c9urm9887cmldlsa3w3jhxe0knfy5

```
*** Please sign add validator hash on the ledger device ***
```

After that, the command shows the connected Ledger's address, and asks the user to sign the TX with the Ledger.

```
Partial TX created
```

```
1 of 2 required Add Validator signatures have been signed. Saving TX to disk to enable remaining signing.
```

```
Path to export partially signed TX to:
```

Because you've setup a multisig, TX isn't fully signed, and the commands asks a file to write into. Use something like `partialAddValidatorTx.txt`.

```
Path to export partially signed TX to: partialAddValidatorTx.txt
```

```
Addresses remaining to sign the tx
```

```
P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
```

```
Connect a Ledger with one of the remaining addresses or choose a stored key and run the signing command, or send "partialAddValidatorTx.txt" to another user for signing.
```

```
Signing command:
```

```
avalanche transaction sign testsubnet --input-tx=filepath partialAddValidatorTx.txt
```

Sign and Commit the Add Validator TX

The process is very similar to signing of Subnet Deployment TX. So far, one address has signed the TX, but you need N signatures. To get the remaining signatures, you may connect a different Ledger to the same computer you've been working on. Alternatively, you may send the `partialAddValidatorTx.txt` file to other users to sign themselves.

The remainder of this section assumes that you are working on a machine with access to both the remaining keys and the `partialAddValidatorTx.txt` file.

Issue the Command to Sign the Add Validator TX

Avalanche-CLI can detect the deployment network automatically. For `Mainnet` TXs, it uses your Ledger automatically. For `Fuji Testnet`, the CLI prompts the user to choose the signing mechanism.

```
avalanche transaction sign testsubnet --input-tx=filepath partialAddValidatorTx.txt
```

```
Ledger address: P-avax1g7nkguzg8yju8cq3ndzc9lql2yg69s9ejqa2af
```

```
*** Please sign TX hash on the ledger device ***
```

Next, the command is going to start a new signing process for the Add Validator TX.

This activates a `Please review` window on the Ledger. Navigate to the Ledger's `APPROVE` window by using the Ledger's right button, and then authorize the request by pressing both left and right buttons.

Repeat this processes until all required parties have signed the TX. You should see a message like this:

```
All 2 required Tx signatures have been signed. Saving TX to disk to enable commit.
```

```
Overwriting partialAddValidatorTx.txt
```

```
Tx is fully signed, and ready to be committed
```

```
Commit command:
```

```
avalanche transaction commit testsubnet --input-tx=filepath partialAddValidatorTx.txt
```

Now, `partialAddValidatorTx.txt` contains a fully signed TX.

Issue the Command to Commit the add validator TX

To run submit the fully signed TX, run

```
avalanche transaction commit testsubnet --input-tx=filepath partialAddValidatorTx.txt
```

The CLI recognizes the deployment network automatically and submits the TX appropriately.

```
Transaction successful, transaction ID: K7XNSwcmgjYX7BEdtFB3hEwQc6YFKRq9g7hAUphW4J5bjhEJG
```

You've successfully added the validator to the Subnet.

Avalanche Network Runner

The Avalanche Network Runner (**ANR**) allows a user to define, create and interact with a network of Avalanche nodes. It can be used for development and testing.

Developing P2P systems is hard, and blockchains are no different. A developer can't just focus on the functionality of a node, but needs to consider the dynamics of the network, the interaction of nodes and emergent system properties. A lot of testing can't be addressed by unit testing, but needs a special kind of integration testing, where the code runs in interaction with other nodes, attempting to simulate real network scenarios.

In the context of Avalanche, [Subnets](#) are a special focus which requires new tooling and support for playing, working and testing with this unique feature of the Avalanche ecosystem.

The ANR aims at being a tool for developers and system integrators alike, offering functionality to run networks of AvalancheGo nodes with support for custom node, Subnet and network configurations, allowing to locally test code before deploying to Mainnet or even public testnets like `fuji`.

Note that this tool is not for running production nodes, and that because it is being heavily developed right now, documentation might differ slightly from the actual code.

Installation

```
curl -ssfL https://raw.githubusercontent.com/ava-labs/avalanche-network-runner/main/scripts/install.sh | sh
```

The script installs the binary inside the `~/bin` directory. If the directory doesn't exist, it will be created.

Please make sure that `~/bin` is in your `$PATH`:

```
export PATH=~/bin:$PATH
```

To add it to your path permanently, add an export command to your shell initialization script. If you run `bash`, use `.bashrc`. If you run `zsh`, use `.zshrc`.

Furthermore, `AVALANCHEGO_EXEC_PATH` should be set properly in all shells you run commands related to Avalanche Network Runner. We strongly recommend that you put the following in to your shell's configuration file.

```
# replace execPath with the path to AvalancheGo on your machine
# e.g., ${HOME}/go/src/github.com/ava-labs/avalanchego/build/avalanchego
AVALANCHEGO_EXEC_PATH="${HOME}/go/src/github.com/ava-labs/avalanchego/build/avalanchego"
```

Unless otherwise specified, file paths given below are relative to the root of this repository.

Usage

There are two main ways to use the network-runner:

- Run ANR as a binary

This is the recommended approach for most use cases. Doesn't require Golang installation and provides a RPC server with an HTTP API and a client library for easy interaction.

- Import this repository into your go program

This allows for custom network scenarios and high flexibility, but requires more code to be written.

Running the binary, the user can send requests to the RPC server in order to start a network, create Subnets, add nodes to the network, remove nodes from the network, restart nodes, etc. You can make requests through the `avalanche-network-runner` command or by making API calls. Requests are "translated" into gRPC and sent to the server.

Each node can then also be reached via [API](#) endpoints which each node exposes.

The following diagram is a simplified view of the high level architecture of the tool: 

Examples

When running with the binary, ANR runs a server process as an RPC server which then waits for API calls and handles them. Therefore we run one shell with the RPC server, and another one for issuing calls.

Start the Server

```
avalanche-network-runner server \
--log-level debug \
--port=:8080 \
--grpc-gateway-port=:8081
```

Note that the above command will run until you stop it with `CTRL + C`. Further commands will have to be run in a separate terminal.

The RPC server listens to two ports:

- `port` : the main gRPC port (see [gRPC](#)).
- `grpc-gateway-port` : the gRPC gateway port (see [gRPC-gateway](#)), which allows for HTTP requests.

When using the binary to issue calls, the main port will be hit. In this mode, the binary executes compiled code to issue calls. Alternatively, plain HTTP can be used to issue calls, without the need to use the binary. In this mode, the `grpc-gateway-port` should be queried.

Each of the examples below will show both modes, clarifying its usage.

Run Queries

Ping the Server

```
curl -X POST -k http://localhost:8081/v1/ping -d ''
```

or

```
avalanche-network-runner ping \
--log-level debug \
--endpoint="0.0.0.0:8080"
```

Start a New Avalanche Network with Five Nodes

```
curl -X POST -k http://localhost:8081/v1/control/start -d '{"execPath":"'${AVALANCHEGO_EXEC_PATH}'","numNodes":5}'
```

or

```
avalanche-network-runner control start \
--log-level debug \
--endpoint="0.0.0.0:8080" \
--number-of-nodes=5 \
--avalancheego-path ${AVALANCHEGO_EXEC_PATH}
```

Additional optional parameters which can be passed to the start command:

```
--plugin-dir ${AVALANCHEGO_PLUGIN_PATH} \
--blockchain-specs '[{"vm_name":"subnetevm","genesis":"/tmp/subnet-evm.genesis.json"}]' \
--global-node-config '{"index-enabled":false, "api-admin-enabled":true,"network-peer-list-gossip-frequency":"300ms"}' \
--custom-node-configs '{"node1":{"log-level":"debug","api-admin-enabled":false}, "node2":{...}, ...}'
```

`--plugin-dir` and `--blockchain-specs` are parameters relevant to Subnet operation.

`--plugin-dir` can be used to indicate to ANR where it will find plugin binaries for your own VMs. It is optional. If not set, ANR will assume a default location which is relative to the `avalancheego-path` given.

`--blockchain-specs` specifies details about how to create your own blockchains. It takes a JSON array for each blockchain, with the following possible fields:

```
"vm_name": human readable name for the VM
"genesis": path to a file containing the genesis for your blockchain (must be a valid path)
```

See the [Avalanche-CLI documentation](#) for details about how to create and run Subnets with our `Avalanche-CLI` tool.

The network-runner supports AvalancheGo node configuration at different levels.

1. If neither `--global-node-config` nor `--custom-node-configs` is supplied, all nodes get a standard set of config options. Currently this set contains:

```
{
  "network-peer-list-gossip-frequency": "250ms",
  "network-max-reconnect-delay": "1s",
  "public-ip": "127.0.0.1",
  "health-check-frequency": "2s",
  "api-admin-enabled": true,
  "api-ipcs-enabled": true,
  "index-enabled": true
}
```

2. `--global-node-config` is a JSON string representing a *single* AvalancheGo config, which will be applied to **all nodes**. This makes it easy to define common properties to all nodes. Whatever is set here will be *combined* with the standard set above.
3. `--custom-node-configs` is a map of JSON strings representing the *complete* network with individual configs. This allows to configure each node independently. If set, `--number-of-nodes` will be **ignored** to avoid conflicts.
4. The configs can be combined and will be merged, that is one could set global `--global-node-config` entries applied to each node, and also set `--custom-node-configs` for additional entries.
5. Common `--custom-node-configs` entries override `--global-node-config` entries which override the standard set.
6. The following entries will be **ignored in all cases** because the network-runner needs to set them internally to function properly:

```
--log-dir  
--db-dir  
--http-port  
--staking-port  
--public-ipc
```

Wait for All the Nodes in the Cluster to Become Healthy

```
curl -X POST -k http://localhost:8081/v1/control/health -d ''
```

or

```
avalanche-network-runner control health \  
--log-level debug \  
--endpoint="0.0.0.0:8080"
```

The response to this call is actually pretty large, as it contains the state of the whole cluster. At the very end of it there should be a text saying `healthy:true` (it would say `false` if it wasn't healthy).

Get API Endpoints of All Nodes in the Cluster

```
curl -X POST -k http://localhost:8081/v1/control/uris -d ''
```

or

```
avalanche-network-runner control uris \  
--log-level debug \  
--endpoint="0.0.0.0:8080"
```

Query Cluster Status from the Server

```
curl -X POST -k http://localhost:8081/v1/control/status -d ''
```

or

```
avalanche-network-runner control status \  
--log-level debug \  
--endpoint="0.0.0.0:8080"
```

Stream Cluster Status

```
avalanche-network-runner control \  
--request-timeout=3m \  
stream-status \  
--push-interval=5s \  
--log-level debug \  
--endpoint="0.0.0.0:8080"
```

Remove (Stop) a Node

```
curl -X POST -k http://localhost:8081/v1/control/removenode -d '{"name":"node5"}'
```

or

```
avalanche-network-runner control remove-node node5 \  
--request-timeout=3m \  
--log-level debug \  
--endpoint="0.0.0.0:8080" \  
--
```

Restart a Node

In this example we are stopping the node named `node1`.

Note: By convention all node names start with `node` and a number. We suggest to stick to this convention to avoid issues.

```
# e.g., ${HOME}/go/src/github.com/ava-labs/avalanchego/build/avalanchego  
AVALANCHEGO_EXEC_PATH="avalanchego"
```

Note that you can restart the node with a different binary by providing

```
curl -X POST -k http://localhost:8081/v1/control/restartnode -d '{"name":"node1","execPath":"'${AVALANCHEGO_EXEC_PATH}'"}'
```

or

```
avalanche-network-runner control restart-node node1 \
--request-timeout=3m \
--log-level debug \
--endpoint="0.0.0.0:8080" \
--avalanche-go-path ${AVALANCHEGO_EXEC_PATH}
```

Add a Node

In this example we are adding a node named `node99`.

```
# e.g., ${HOME}/go/src/github.com/ava-labs/avalanche/go/build/avalanche \
AVALANCHEGO_EXEC_PATH="avalanche"
```

Note that you can add the new node with a different binary by providing

```
curl -X POST -k http://localhost:8081/v1/control/addnode -d '{"name":"node99","execPath":"'${AVALANCHEGO_EXEC_PATH}'"}'
```

or

```
avalanche-network-runner control add-node node99 \
--request-timeout=3m \
--endpoint="0.0.0.0:8080" \
--avalanche-go-path ${AVALANCHEGO_EXEC_PATH}
```

It's also possible to provide individual node config parameters:

```
--node-config '{"index-enabled":false, "api-admin-enabled":true,"network-peer-list-gossip-frequency":"300ms"}'
```

`--node-config` allows to specify specific AvalancheGo config parameters to the new node. See [here](#) for the reference of supported flags.

Note: The following parameters will be *ignored* if set in `--node-config`, because the network runner needs to set its own in order to function properly: `--log-dir` `--db-dir`

Note: The following Subnet parameters will be set from the global network configuration to this node: `--track-subnets` `--plugin-dir`

Terminate the Cluster

Note that this will still require to stop your RPC server process with `Ctrl+C` to free the shell.

```
curl -X POST -k http://localhost:8081/v1/control/stop -d ''
```

or

```
avalanche-network-runner control stop \
--log-level debug \
--endpoint="0.0.0.0:8080"
```

Subnets

For general Subnet documentation, please refer to [Subnets](#). ANR can be a great helper working with Subnets, and can be used to develop and test new Subnets before deploying them in public networks. However, for a smooth and guided experience, we recommend using [Avalanche-CLI](#). These examples expect a basic understanding of what Subnets are and their usage.

RPC Server Subnet-EVM Example

The Subnet-EVM is a simplified version of Coreth VM (C-Chain). This chain implements the Ethereum Virtual Machine and supports Solidity smart-contracts as well as most other Ethereum client functionality. It can be used to create your own fully Ethereum-compatible Subnet running on Avalanche. This means you can run your Ethereum-compatible dApps in custom Subnets, defining your own gas limits and fees, and deploying solidity smart-contracts while taking advantage of Avalanche's validator network, fast finality, consensus mechanism and other features. Essentially, think of it as your own Ethereum where you can concentrate on your business case rather than the infrastructure. See [Subnet-EVM](#) for further information.

Using Avalanche Network as a Library

The Avalanche Network Runner can also be imported as a library into your programs so that you can use it to programmatically start, interact with and stop Avalanche networks. For an example of using the Network Runner in a program, see an [example](#).

Creating a network is as simple as:

```
network, err := local.NewDefaultNetwork(log, binaryPath)
```

where `log` is a logger of type `logging.Logger` and `binaryPath` is the path of the AvalancheGo binary that each node that exists on network startup will run.

For example, the below snippet creates a new network using default configurations, and each node in the network runs the binaries at `/home/user/go/src/github.com/ava-labs/avalanchego/build`:

```
network, err := local.NewDefaultNetwork(log, "/home/user/go/src/github.com/ava-labs/avalanchego/build")
```

Once you create a network, you must eventually call `Stop()` on it to make sure all of the nodes in the network stop. Calling this method kills all of the Avalanche nodes in the network. You probably want to call this method in a `defer` statement to make sure it runs.

To wait until the network is ready to use, use the network's `Healthy` method. It returns a channel which will be notified when all nodes are healthy.

Each node has a unique name. Use the network's `GetNodeNames()` method to get the names of all nodes.

Use the network's method `GetNode(string)` to get a node by its name. For example:

```
names, _ := network.GetNodeNames()
node, _ := network.GetNode(names[0])
```

Then you can make API calls to the node:

```
id, _ := node.GetAPIClient().InfoAPI().GetNodeID() // Gets the node's node ID
balance, _ := node.GetAPIClient().XChainAPI().GetBalance(address, assetID, false) // Pretend these arguments are defined
```

After a network has been created and is healthy, you can add or remove nodes to/from the network:

```
newNode, _ := network.AddNode(nodeConfig)
err := network.RemoveNode(names[0])
```

Where `nodeConfig` is a struct which contains information about the new node to be created. For a local node, the most important elements are its name, its binary path and its identity, given by a TLS key/cert.

You can create a network where nodes are running different binaries -- just provide different binary paths to each:

```
stakingCert, stakingKey, err := staking.NewCertAndKeyBytes()
if err != nil {
    return err
}
nodeConfig := node.Config{
    Name: "New Node",
    ImplSpecificConfig: local.NodeConfig{
        BinaryPath: "/tmp/my-avalanchego/build",
    },
    StakingKey: stakingKey,
    StakingCert: stakingCert,
}
```

After adding a node, you may want to call the network's `Healthy` method again and wait until the new node is healthy before making API calls to it.

Creating Custom Networks

To create custom networks, pass a custom config (the second parameter) to the `local.NewNetwork(logging.Logger, network.Config)` function. The config defines the number of nodes when the network starts, the genesis state of the network, and the configs for each node.

Please refer to [NetworkConfig](#) for more details.

Avalanche-CLI Commands

Avalanche-CLI is a command-line tool that gives developers access to everything Avalanche. This release specializes in helping developers build and test Subnets.

To get started, look at the documentation for the subcommands or jump right in with `avalanche subnet create myNewSubnet`.

Subnet

The `subnet` command suite provides a collection of tools for developing and deploying Subnets.

To get started, use the `subnet create` command wizard to walk through the configuration of your very first Subnet. Then, go ahead and deploy it with the `subnet deploy` command. You can use the rest of the commands to manage your Subnet configurations and live deployments.

Subnet AddValidator

The `subnet addValidator` command whitelists a primary network validator to validate the provided deployed Subnet.

To add the validator to the Subnet's allow list, you first need to provide the subnetName and the validator's unique NodeID. The command then prompts for the validation start time, duration, and stake weight. You can bypass these prompts by providing the values with flags.

This command currently only works on Subnets deployed to either the Fuji Testnet or Mainnet.

Usage:

```
avalanche subnet addValidator [subnetName] [flags]
```

Flags:

--fuji	fuji	join on fuji (alias for `testnet`)
-h, --help		help for addValidator
-k, --key string		select the key to use [fuji deploy only]
-g, --ledger		use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs strings		use the given ledger addresses
--mainnet	mainnet	join on mainnet
--nodeID string		set the NodeID of the validator to add
--output-tx-path string		file path of the add validator tx
--staking-period duration		how long this validator will be staking
--start-time string		UTC start time when this validator starts validating, in 'YYYY-MM-DD HH:MM:SS' format
--subnet-auth-keys strings		control keys that will be used to authenticate add validator tx
--testnet	testnet	join on testnet (alias for `fuji`)
--weight uint		set the staking weight of the validator to add

Remove Validator in a Subnet

This command removes a node as a validator in a Subnet.

Usage:

```
avalanche subnet removeValidator [subnetName] [flags]
```

Flags:

--fuji		remove validator in existing fuji deployment (alias for `testnet`)
-k, --key	string	select the key to use [fuji only]
-g, --ledger		use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs	strings	use the given ledger addresses
--local		remove validator in existing local deployment
--mainnet		remove validator in existing mainnet deployment
--nodeID	string	node id of node to be added as validator in elastic subnet
--output-tx-path	string	file path of the removeValidator tx
--subnet-auth-keys	strings	control keys that will be used to authenticate removeValidator tx
--testnet		remove validator in existing testnet deployment (alias for `fuji`)

Subnet Configure

AvalancheGo nodes support several different configuration files. Subnets have their own Subnet config which applies to all chains/VMs in the Subnet. Each chain within the Subnet can have its own chain config. This command allows you to set both config files.

Usage:

```
avalanche subnet configure [subnetName] [flags]
```

Flags:

--chain-config	string	path to the chain configuration
-h, --help		help for configure
--per-node-chain-config	string	path to per node chain configuration for local network
--subnet-config	string	path to the subnet configuration

Subnet Create

The `subnet create` command builds a new genesis file to configure your Subnet. By default, the command runs an interactive wizard. It walks you through all the steps you need to create your first Subnet.

The tool supports deploying Subnet-EVM and custom VMs. You can create a custom, user-generated genesis with a custom VM by providing the path to your genesis and VM binaries with the `--genesis` and `--vm` flags.

By default, running the command with a `subnetName` that already exists causes the command to fail. If you'd like to overwrite an existing configuration, pass the `-f` flag.

Usage:

```
avalanche subnet create [subnetName] [flags]
```

Flags:

```
--custom          use a custom VM template
--evm            use the SubnetEVM as the base template
-f, --force       overwrite the existing configuration if one exists
--genesis string  file path of genesis to use
-h, --help        help for create
--latest         use latest VM version, takes precedence over --vm-version
--vm string      file path of custom vm to use
--vm-version string  version of vm template to use
```

Subnet Delete

The `subnet delete` command deletes an existing Subnet configuration.

Usage:

```
avalanche subnet delete [flags]
```

Flags:

```
-h, --help help for delete
```

Subnet Deploy

The `subnet deploy` command deploys your Subnet configuration locally, to Fuji Testnet, or to Mainnet.

At the end of the call, the command prints the RPC URL you can use to interact with the Subnet.

Avalanche-CLI only supports deploying an individual Subnet once per network. Subsequent attempts to deploy the same Subnet to the same network (local, Fuji, Mainnet) aren't allowed. If you'd like to redeploy a Subnet locally for testing, you must first call [avalanche network clean](#) to reset all deployed chain state. Subsequent local deploys redeploy the chain with fresh state. You can deploy the same Subnet to multiple networks, so you can take your locally tested Subnet and deploy it on Fuji or Mainnet.

Usage:

```
avalanche subnet deploy [subnetName] [flags]
```

Flags:

```
--avalanchego-version string  use this version of avalanchego (ex: v1.17.12) (default "latest")
--control-keys strings        addresses that may make subnet changes
-f, --fuji testnet           deploy to fuji (alias to testnet)
-h, --help                    help for deploy
-k, --key string             select the key to use [fuji deploy only]
-g, --ledger                  use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs strings       use the given ledger addresses
-l, --local                   deploy to a local network
-m, --mainnet                 deploy to mainnet
--output-tx-path string      file path of the blockchain creation tx
-s, --same-control-key       use creation key as control key
--subnet-auth-keys strings   control keys that will be used to authenticate chain creation
-t, --testnet fuji            deploy to testnet (alias to fuji)
--threshold uint32            required number of control key signatures to make subnet changes
```

Subnet Describe

The `subnet describe` command prints the details of a Subnet configuration to the console. By default, the command prints a summary of the configuration. By providing the `--genesis` flag, the command instead prints out the raw genesis file.

Usage:

```
avalanche subnet describe [subnetName] [flags]
```

Flags:

```
-g, --genesis  Print the genesis to the console directly instead of the summary
-h, --help      help for describe
```

Subnet Export

The `subnet export` command write the details of an existing Subnet deploy to a file.

The command prompts for an output path. You can also provide one with the `--output` flag.

Usage:

```
avalanche subnet export [subnetName] [flags]
```

Flags:

```
-h, --help      help for export
-o, --output string  write the export data to the provided file path
```

Subnet Import

The `subnet import` command imports configurations into Avalanche-CLI.

This command supports importing from a file created on another computer, or importing from Subnets running public networks (for example, created manually or with the deprecated Subnet-CLI)

Import from a File

To import from a file, you can optionally provide the path as a command-line argument. Alternatively, running the command without any arguments triggers an interactive wizard. To import from a repository, go through the wizard. By default, an imported Subnet doesn't overwrite an existing Subnet with the same name. To allow overwrites, provide the `--force` flag.

Usage:

```
avalanche subnet import file [subnetPath] [flags]
```

Flags:

```
--branch string  the repo branch to use if downloading a new repo
-f, --force       overwrite the existing configuration if one exists
-h, --help        help for import
--repo string    the repo to import (ex: ava-labs/avalanche-plugins-core) or url to download the repo from
--subnet string  the subnet configuration to import from the provided repo
```

Import from a Public Network

The `subnet import public` command imports a Subnet configuration from a running network.

The genesis file should be available from the disk for this to work. By default, an imported Subnet doesn't overwrite an existing Subnet with the same name. To allow overwrites, provide the `--force` flag.

Usage:

```
avalanche subnet import public [subnetPath] [flags]
```

Flags:

```
--custom          use a custom VM template
--evm             import a subnet-evm
-f, --force        overwrite the existing configuration if one exists
--fuji fuji       import from fuji (alias for `testnet`)
--genesis-file-path string  path to the genesis file
-h, --help         help for public
--mainnet mainnet import from mainnet
--node-url string [optional] URL of an already running subnet validator
--subnet-id string the subnet ID
--testnet testnet import from testnet (alias for `fuji`)
```

Subnet Join

The `subnet join` command configures your validator node to begin validating a new Subnet.

To complete this process, you must have access to the machine running your validator. If the CLI is running on the same machine as your validator, it can generate or update your node's config file automatically. Alternatively, the command can print the necessary instructions to update your node manually. To complete the validation process, the Subnet's admins must add the NodeID of your validator to the Subnet's allow list by calling `addValidator` with your NodeID.

After you update your validator's config, you need to restart your validator manually. If you provide the `--avalanchego-config` flag, this command attempts to edit the config file at that path.

This command currently only supports Subnets deployed on the Fuji Testnet and Mainnet.

Usage:

```
avalanche subnet join [subnetName] [flags]
```

Flags:

```
--avalanchego-config string      file path of the avalanchego config file
--fail-if-not-validating        fail if whitelist check fails
--force-whitelist-check         if true, force the whitelist check
--force-write                   if true, skip to prompt to overwrite the config file
--fuji fuji                     join on fuji (alias for `testnet`)
-h, --help                      help for join
--mainnet mainnet                join on mainnet
--nodeID string                 set the NodeID of the validator to check
--plugin-dir string              file path of avalanchego's plugin directory
--print                          if true, print the manual config without prompting
--skip-whitelist-check          if true, skip the whitelist check
--testnet testnet                join on testnet (alias for `fuji`)
```

Subnet List

The `subnet list` command prints the names of all created Subnet configurations. Without any flags, it prints some general, static information about the Subnet. With the `--deployed` flag, the command shows additional information including the VMID, BlockchainID and SubnetID.

Usage:

```
 avalanche subnet list [flags]
```

Flags:

```
--deployed    show additional deploy information
-h, --help      help for list
```

Subnet Publish

The `subnet publish` command publishes the Subnet's VM to a repository.

Usage:

```
 avalanche subnet publish [subnetName] [flags]
```

Flags:

```
--alias string           We publish to a remote repo, but identify the repo locally under a user-provided alias (e.g.
myrepo).
--force                  If true, ignores if the subnet has been published in the past, and attempts a forced publish.
-h, --help                help for publish
--no-repo-path string    Do not let the tool manage file publishing, but have it only generate the files and put them in
the location given by this flag.
--repo-url string         The URL of the repo where we are publishing
--subnet-file-path string Path to the Subnet description file. If not given, a prompting sequence will be initiated.
--vm-file-path string     Path to the VM description file. If not given, a prompting sequence will be initiated.
```

Subnet Stats

The `subnet stats` command prints validator statistics for the given Subnet.

Usage:

```
 avalanche subnet stats [subnetName] [flags]
```

Flags:

```
--fuji fuji            print stats on fuji (alias for `testnet`)
-h, --help              help for stats
--mainnet mainnet       print stats on mainnet
--testnet testnet       print stats on testnet (alias for `fuji`)
```

Subnet VMID

The `subnet vmid` command prints the virtual machine ID (VMID) for the given Subnet.

Usage:

```
 avalanche subnet vmid [subnetName]
```

Elastic Subnet

Transforms permissioned Subnet into Elastic Subnet

This command transforms your permissioned Subnet into an Elastic Subnet (NOTE: this action is irreversible).

Usage:

```
avalanche subnet elastic [subnetName] [flags]
```

Flags:

--default		If true, use default elastic subnet config values
--denomination	int	specify the token denomination (for Fuji and Mainnet only)
--force		If true, override transform into elastic subnet warning
--fuji		elastic subnet transform existing fuji deployment (alias for `testnet`)
-k, --key	string	select the key to use [fuji only]
-g, --ledger		use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs	strings	use the given ledger addresses
--local		elastic subnet transform existing local deployment
--mainnet		elastic subnet transform existing mainnet deployment
--output-tx-path	string	file path of the transformSubnet tx
--stake-amount	int	amount of tokens to stake on validator
--staking-period	string	how long validator validates for after start time
--start-time	string	when validator starts validating (format as "2006-01-02 15:00:00")
--subnet-auth-keys	strings	control keys that will be used to authenticate transformSubnet tx
--testnet		elastic subnet transform existing testnet deployment (alias for `fuji`)
--tokenName	string	specify the token name
--tokenSymbol	string	specify the token symbol
--transformValidators		If true, transform validators to permissionless validators after elastic transform

Add Permissionless Validator in an Elastic Subnet

This command adds a node as a permissionless validator in an Elastic Subnet.

Usage:

```
avalanche subnet join [subnetName] --elastic [flags]
```

Flags:

--fuji		add permissionless validator in existing fuji deployment (alias for `testnet`)
-k, --key	string	select the key to use [fuji only]
-g, --ledger		use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs	strings	use the given ledger addresses
--local		add permissionless validator in existing local deployment
--mainnet		add permissionless validator in existing mainnet deployment
--nodeID	string	node id of node to be added as validator in elastic subnet
--stake-amount	int	amount of tokens to stake on validator
--staking-period	string	how long validator validates for after start time
--start-time	string	when validator starts validating (format as "2006-01-02 15:00:00")
--testnet		add permissionless validator in existing testnet deployment (alias for `fuji`)

Add Permissionless Delegator in an Elastic Subnet

This command delegates stake to a permissionless validator in an Elastic Subnet.

Usage:

```
avalanche subnet addPermissionlessDelegator [subnetName] [flags]
```

Flags:

--fuji		add permissionless delegator in existing fuji deployment (alias for `testnet`)
-k, --key	string	select the key to use [fuji only]
-g, --ledger		use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs	strings	use the given ledger addresses
--local		add permissionless delegator in existing local deployment
--mainnet		add permissionless delegator in existing mainnet deployment
--nodeID	string	node id of node to be added as validator in elastic subnet
--stake-amount	int	amount of tokens to delegate to validator
--staking-period	string	how long to delegate for after start time
--start-time	string	when to starts delegating (format as "2006-01-02 15:00:00")
--testnet		add permissionless delegator in existing testnet deployment (alias for `fuji`)

Subnet Upgrade

The `subnet upgrade` command suite provides a collection of tools for updating your developmental and deployed Subnets.

Subnet Upgrade Apply

Apply generated upgrade bytes to running Subnet nodes to trigger a network upgrade.

For public networks (Fuji Testnet or Mainnet), to complete this process, you must have access to the machine running your validator. If the CLI is running on the same machine as your validator, it can manipulate your node's configuration automatically. Alternatively, the command can print the necessary instructions to upgrade your node manually.

After you update your validator's configuration, you need to restart your validator manually. If you provide the `--avalanchego-chain-config-dir` flag, this command attempts to write the upgrade file at that path. Refer to [this doc](#) for related documentation.

Usage:

```
 avalanche subnet upgrade apply [subnetName] [flags]
```

Flags:

<code>--avalanchego-chain-config-dir</code> string	avalanchego's chain config file directory (default "/Users/connor/.avalanchego/chains")
<code>--config</code>	create upgrade config for future subnet deployments (same as generate)
<code>--force</code>	If true, don't prompt for confirmation of timestamps in the past
<code>--fuji fuji</code>	apply upgrade existing fuji deployment (alias for `testnet`)
<code>-h, --help</code>	help for apply
<code>--local local</code>	apply upgrade existing local deployment
<code>--mainnet mainnet</code>	apply upgrade existing mainnet deployment
<code>--print</code>	if true, print the manual config without prompting (for public networks only)
<code>--testnet testnet</code>	apply upgrade existing testnet deployment (alias for 'fuji')

Subnet Upgrade Export

Export the upgrade bytes file to a location of choice on disk.

Usage:

```
 avalanche subnet upgrade export [subnetName] [flags]
```

Flags:

<code>--force</code>	If true, overwrite a possibly existing file without prompting
<code>-h, --help</code>	help for export
<code>--upgradefilepath</code> string	Export upgrade bytes file to location of choice on disk

Subnet Upgrade Generate

The `subnet upgrade generate` command builds a new `upgrade.json` file to customize your Subnet. It guides the user through the process using an interactive wizard.

Usage:

```
 avalanche subnet upgrade generate [subnetName] [flags]
```

Flags:

<code>-h, --help</code>	help for generate
-------------------------	-------------------

Subnet Upgrade Import

Import the upgrade bytes file into the local environment.

Usage:

```
 avalanche subnet upgrade import [subnetName] [flags]
```

Flags:

<code>-h, --help</code>	help for import
<code>--upgradefilepath</code> string	Import upgrade bytes file into local environment

Subnet Upgrade Print

Print the `upgrade.json` file content.

Usage:

```
 avalanche subnet upgrade print [subnetName] [flags]
```

Flags:

```
-h, --help      help for list
```

Subnet Upgrade VM

The `subnet upgrade vm` command enables the user to upgrade their Subnet's VM binary. The command can upgrade both local Subnets and publicly deployed Subnets on Fuji and Mainnet.

The command walks the user through an interactive wizard. The user can skip the wizard by providing command line flags.

Usage:

```
avalanche subnet upgrade export [subnetName] [flags]
```

Flags:

```
--deployed    show additional deploy information  
-h, --help      help for list
```

Network

The `network` command suite provides a collection of tools for managing local Subnet deployments.

When you deploy a Subnet locally, it runs on a local, multi-node Avalanche network. The `subnet deploy` command starts this network in the background. This command suite allows you to shutdown, restart, and clear that network.

This network currently supports multiple, concurrently deployed Subnets.

Network Clean

The `network clean` command shuts down your local, multi-node network. All deployed Subnets shutdown and delete their state. You can restart the network by deploying a new Subnet configuration.

Usage:

```
avalanche network clean [flags]
```

Flags:

```
--hard      Also clean downloaded avanchego and plugin binaries  
-h, --help      help for clean
```

Network Start

The `network start` command starts a local, multi-node Avalanche network on your machine.

By default, the command loads the default snapshot. If you provide the `--snapshot-name` flag, the network loads that snapshot instead. The command fails if the local network is already running.

Usage:

```
avalanche network start [flags]
```

Flags:

```
--avanchego-version string    use this version of avanchego (ex: v1.17.12) (default "latest")  
-h, --help                      help for start  
--snapshot-name string          name of snapshot to use to start the network from (default "default-1654102509")
```

Network Status

The `network status` command prints whether or not a local Avalanche network is running and some basic stats about the network.

Usage:

```
avalanche network status [flags]
```

Flags:

```
-h, --help      help for status
```

Network Stop

The `network stop` command shuts down your local, multi-node network.

All deployed Subnets shutdown gracefully and save their state. If you provide the `--snapshot-name` flag, the network saves its state under this named snapshot. You can reload this snapshot with `network start --snapshot-name <snapshotName>`. Otherwise, the network saves to the default snapshot, overwriting any existing state. You can reload the default snapshot with `network start`.

Usage:

```
avalanche network stop [flags]
```

Flags:

```
-h, --help           help for stop
--snapshot-name string   name of snapshot to use to save network state into (default "default-1654102509")
```

Transaction

The `transaction` command suite provides all of the utilities required to sign multisig transactions.

Transaction Commit

The `transaction commit` command commits a transaction by submitting it to the P-Chain.

Usage:

```
avalanche transaction commit [subnetName] [flags]
```

Flags:

```
-h, --help           help for commit
--input-txfilepath string   Path to the transaction signed by all signatories
```

Transaction Sign

The `transaction sign` command signs a multisig transaction.

Usage:

```
avalanche transaction sign [subnetName] [flags]
```

Flags:

```
-h, --help           help for sign
--input-txfilepath string   Path to the transaction file for signing
-k, --key string      select the key to use [fuji only]
-g, --ledger          use ledger instead of key (always true on mainnet, defaults to false on fuji)
--ledger-addrs strings   use the given ledger addresses
```

Key

The `key` command suite provides a collection of tools for creating and managing signing keys. You can use these keys to deploy Subnets to the Fuji Testnet, but these keys are NOT suitable to use in production environments. DO NOT use these keys on Mainnet.

To get started, use the `key create` command.

Key Create

The `key create` command generates a new private key to use for creating and controlling test Subnets. Keys generated by this command are NOT cryptographically secure enough to use in production environments. DO NOT use these keys on Mainnet.

The command works by generating a secp256 key and storing it with the provided `keyName`. You can use this key in other commands by providing this `keyName`.

If you'd like to import an existing key instead of generating one from scratch, provide the `--file` flag.

Usage:

```
avalanche key create [keyName] [flags]
```

Flags:

```
--file string      import the key from an existing key file
-f, --force         overwrite an existing key with the same name
-h, --help          help for create
```

Key Delete

The `key delete` command deletes an existing signing key.

To delete a key, provide the `keyName`. The command prompts for confirmation before deleting the key. To skip the confirmation, provide the `--force` flag.

Usage:

```
avalanche key delete [keyName] [flags]
```

Flags:

```
-f, --force    delete the key without confirmation
-h, --help     help for delete
```

Key Export

The `key export` command exports a created signing key. You can use an exported key in other applications or import it into another instance of Avalanche-CLI.

By default, the tool writes the hex encoded key to stdout. If you provide the `--output` flag, the command writes the key to a file of your choosing.

Usage:

```
avalanche key export [keyName] [flags]
```

Flags:

```
-h, --help     help for export
-o, --output string  write the key to the provided file path
```

Key List

The `key list` command prints information for all stored signing keys or for the ledger addresses associated to certain indices.

Usage:

```
avalanche key list [flags]
```

Flags:

```
-a, --all-networks  list all network addresses
-c, --cchain        list C-Chain addresses (default true)
-f, --fuji          list testnet (fuji) network addresses
-h, --help          help for list
-g, --ledger uints list ledger addresses for the given indices (default [])
-l, --local          list local network addresses
-m, --mainnet        list mainnet network addresses
-t, --testnet        list testnet (fuji) network addresses
```

Elastic Subnets Parameters

Avalanche Permissioned Subnets can be turned into Elastic Subnets via the [TransformSubnetTx](#) transaction. `TransformSubnetTx` specifies a set of structural parameters for the Elastic Subnet. This reference describes these structural parameters and illustrates the constraints they must satisfy.

Elastic Subnet Parameters

Subnet

`Subnet` has type `ids.ID` and it's the Subnet ID. `Subnet` is the ID of the `CreateSubnetTx` transaction that created the Subnet in the first place. The following constraints apply:

- `Subnet` must be different from `PrimaryNetworkID`.

AssetID

`AssetID` has type `ids.ID` and it's the ID of the asset to use when staking on the Subnet. The following constraints apply:

- `AssetID` must not be the `Empty ID`.
- `AssetID` must not be `AVAX ID`, the Primary Network asset.

InitialSupply

`InitialSupply` has type `uint64` and it's the initial amount of `AssetID` transferred in the Elastic Subnet upon its transformation. Such amount is available for distributing staking rewards. The following constraints apply:

- `InitialSupply` must be larger than zero.

MaximumSupply

`MaximumSupply` has type `uint64` and it's the maximum amount of `AssetID` that Subnet has available for staking and rewards at any time. The following constraints apply:

- `MaximumSupply` must be larger or equal to `InitialSupply`.

A Subnet supply can vary in time but it should be no larger than the configured maximum at any point in time, including at Subnet creation.

MinConsumptionRate

`MinConsumptionRate` has type `uint64` and it's the minimal rate to allocate funds to validator rewards. You can find more details about it in the [Reward Formula section](#). The following constraints apply:

- `MinConsumptionRate` must be smaller or equal to `PercentDenominator`.

See [Notes on Percentages](#) section to understand `PercentDenominator` role.

MaxConsumptionRate

`MaxConsumptionRate` has type `uint64` and it's the maximal rate to allocate funds to validator rewards. You can find more details about it in the [Reward Formula section](#). The following constraints apply:

- `MaxConsumptionRate` must be larger or equal to `MinConsumptionRate`.
- `MaxConsumptionRate` must be smaller or equal to `PercentDenominator`.

See [Notes on Percentages](#) section to understand `PercentDenominator` role.

MinValidatorStake

`MinValidatorStake` has type `uint64` and it's the minimum amount of funds required to become a validator. The following constraints apply:

- `MinValidatorStake` must be larger than zero
- `MinValidatorStake` must be smaller or equal to `InitialSupply`

MaxValidatorStake

`MaxValidatorStake` has type `uint64` and it's the maximum amount of funds a single validator can be allocated, including delegated funds. The following constraints apply:

- `MaxValidatorStake` must be larger or equal to `MinValidatorStake`
- `MaxValidatorStake` must be smaller or equal to `MaximumSupply`

MinStakeDuration

`MinStakeDuration` has type `uint32` and it's the minimum number of seconds a staker can stake for. The following constraints apply:

- `MinStakeDuration` must be larger than zero.

MaxStakeDuration

`MaxStakeDuration` has type `uint32` and it's the maximum number of seconds a staker can stake for. The following constraints apply:

- `MaxStakeDuration` must be larger or equal to `MinStakeDuration`.
- `MaxStakeDuration` must be smaller or equal to `GlobalMaxStakeDuration`.

`GlobalMaxStakeDuration` is defined in genesis and applies to both the Primary Network and all Subnets.

Its Mainnet value is \$365 |times 24 |times time.Hour\$.

MinDelegationFee

`MinDelegationFee` has type `uint32` and it's the minimum fee rate a delegator must pay to its validator for delegating. `MinDelegationFee` is a percentage; the actual fee is calculated multiplying the fee rate for the delegator reward. The following constraints apply:

- `MinDelegationFee` must be smaller or equal to `PercentDenominator`.

The `MinDelegationFee` rate applies to Primary Network as well. Its Mainnet value is \$2%\$.

MinDelegatorStake

`MinDelegatorStake` has type `uint64` and it's the minimum amount of funds required to become a delegator. The following constraints apply:

- `MinDelegatorStake` must be larger than zero.

MaxValidatorWeightFactor

`MaxValidatorWeightFactor` has type `uint8` and it's the factor which calculates the maximum amount of delegation a validator can receive. A value of 1 effectively disables delegation. You can find more details about it in the [Delegators Weight Checks section](#). The following constraints apply:

- `MaxValidatorWeightFactor` must be larger than zero.

UptimeRequirement

`UptimeRequirement` has type `uint32` and it's the minimum percentage of its staking time that a validator must be online and responsive for to receive a reward. The following constraints apply:

- `UptimeRequirement` must be smaller or equal `PercentDenominator`.

See [Notes on Percentages](#) section to understand `PercentDenominator` role.

Reward Formula

Consider an Elastic Subnet validator which stakes a `$Stake$` amount `AssetID` for `$StakingPeriod$` seconds.

Assume that at the start of the staking period there is a `$Supply$` amount of `AssetID` in the Subnet. The maximum amount of Subnet asset is `$MaximumSupply$ AssetID`.

Then at the end of its staking period, a responsive Elastic Subnet validator receives a reward calculated as follows:

$$\text{Reward} = (\text{MaximumSupply} - \text{Supply}) \times \frac{\text{Stake}}{\text{Supply}} \times \frac{\text{Staking Period}}{\text{Minting Period}} \times \text{EffectiveConsumptionRate}$$

where $\text{EffectiveConsumptionRate} = \frac{\text{MinConsumptionRate} \times \text{PercentDenominator}}{\text{MaxConsumptionRate} \times \text{PercentDenominator}}$

Note that `$StakingPeriod$` is the staker's entire staking period, not just the staker's uptime, that is the aggregated time during which the staker has been responsive. The uptime comes into play only to decide whether a staker should be rewarded; to calculate the actual reward only the staking period duration is taken into account.

`$EffectiveConsumptionRate$` is a linear combination of `$MinConsumptionRate$` and `$MaxConsumptionRate$`. `$MinConsumptionRate$` and `$MaxConsumptionRate$` bound `$EffectiveConsumptionRate$` because

$$\text{MinConsumptionRate} \leq \text{EffectiveConsumptionRate} \leq \text{MaxConsumptionRate}$$

The larger `$StakingPeriod$` is, the closer `$EffectiveConsumptionRate$` is to `$MaxConsumptionRate$`.

A staker achieves the maximum reward for its stake if `$StakingPeriod$` = `$Minting Period$`. The reward is:

$$\text{Max Reward} = (\text{MaximumSupply} - \text{Supply}) \times \frac{\text{Stake}}{\text{Supply}} \times \frac{\text{MaxConsumptionRate}}{\text{PercentDenominator}}$$

Note that the reward formula above is used in the Primary Network to calculate stakers reward. For reference, you can find Primary network parameters in [the section below](#).

Delegators Weight Checks

There are bounds set of the maximum amount of delegators' stake that a validator can receive.

The maximum weight `$MaxWeight$` a validator `$Validator$` can have is:

$$\text{MaxWeight} = \min(\text{Validator.Weight} \times \text{MaxValidatorWeightFactor}, \text{MaxValidatorStake})$$

where `$MaxValidatorWeightFactor$` and `$MaxValidatorStake$` are the Elastic Subnet Parameters described above.

A delegator won't be added to a validator if the combination of their weights and all other validator's delegators' weight is larger than `$MaxWeight$`. Note that this must be true at any point in time.

Note that setting `$MaxValidatorWeightFactor$` to 1 disables delegation since the `$MaxWeight = Validator.Weight$`.

Notes on Percentages

`PercentDenominator = 1_000_000` is the denominator used to calculate percentages.

It allows you to specify percentages up to 4 digital positions. To denominate your percentage in `PercentDenominator` just multiply it by `10_000`. For example:

- `100%` corresponds to $100 * 10_000 = 1_000_000$
- `1%` corresponds to $1 * 10_000 = 10_000$
- `0.02%` corresponds to $0.002 * 10_000 = 200$
- `0.0007%` corresponds to $0.0007 * 10_000 = 7$

Primary Network Parameters on Mainnet

An Elastic Subnet is free to pick any parameters affecting rewards, within the constraints specified above. For reference we list below Primary Network parameters on Mainnet:

- `AssetID = Avax`
- `InitialSupply = 240_000_000 Avax`
- `MaximumSupply = 720_000_000 Avax`.
- `MinConsumptionRate = 0.10 * reward.PercentDenominator`.
- `MaxConsumptionRate = 0.12 * reward.PercentDenominator`.
- `Minting Period = 365 * 24 * time.Hour`.
- `MinValidatorStake = 2_000 Avax`.
- `MaxValidatorStake = 3_000_000 Avax`.
- `MinStakeDuration = 2 * 7 * 24 * time.Hour`.
- `MaxStakeDuration = 365 * 24 * time.Hour`.
- `MinDelegationFee = 20000`, that is `2%`.
- `MinDelegatorStake = 25 Avax`.
- `MaxValidatorWeightFactor = 5`. This is a platformVM parameter rather than a genesis one, so it's shared across networks.
- `UptimeRequirement = 0.8`, that is `80%`.

Run a Mainnet Node on DeFi Kingdoms Subnet

Introduction

This article describes how to run a Mainnet node on [DeFi Kingdoms \(DFK\) Subnet](#). It can be applied to any other Subnet, where the corresponding part of the Subnet info should be replaced.

Following necessary steps are needed to run your node on the DFK Subnet:

1. Build the AvalancheGo binary
2. Build the plugin binary for the DFK Subnet-EVM
3. Track the DFK Subnet
4. Connect to the DFK Subnet!

Just want the commands? Jump to the [end!](#)

Build AvalancheGo Binary

First, you need to download and build AvalancheGo (handles the orchestration of running Custom VMs). You can follow [this comprehensive guide](#) to complete this step. For this tutorial, we recommend compiling from source instead of using the `AvalancheGo Installer`.

Build subnet-evm Binary

For the steps below, we will assume that you completed first step successfully and are now in your AvalancheGo directory (within your `$GOPATH`).

Next, you will clone the DFK Subnet-EVM repository:

```
mkdir -p $GOPATH/src/github.com/ava-labs
cd $GOPATH/src/github.com/ava-labs
git clone https://github.com/ava-labs/subnet-evm.git
cd subnet-evm
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/subnet-evm.git
```

You can find more about SSH and how to use it [here](#).

Now that you are in the `ava-labs/subnet-evm` repository, you will build the binary and place it directly into the `plugins` directory. To do this, you will pass in the desired path to place the plugin binary. You will want to place this binary into the `plugins` directory of AvalancheGo.

```
./scripts/build.sh ~/.avalanchego/plugins/mDV3QWRXfwgKUWb9sggkv4vQxAQR4y2CyKrt5pLZ5SzQ7EHBv
```

The long string `mDV3QWRXfwgKUWb9sggkv4vQxAQR4y2CyKrt5pLZ5SzQ7EHBv` is the CB58 encoded VMID of the DFK Subnet-EVM. AvalancheGo will use the name of this file to determine what VMs are available to run from the `plugins` directory.

Tracking DFK Subnet and Restarting the Node

AvalancheGo will only validate the primary network by default. In order to add the DFK Subnet, you will need to add the DFK Subnet ID to the set of tracked Subnets in the node's config file or pass it through the command-line options of the node. Once the node's config file has been updated, you will need to start the Avalanche node (restart if already running).

Once you start the node, it will begin syncing the Primary Network. Once the node reaches the point in the Platform Chain where the DFK Subnet is created, it will begin syncing the DFK Subnet as well, and will start validating once it has fully bootstrapped.

Updating Config File

You can skip this section if you want to track Subnets through command-line flags.

You need to create a new config file or edit your existing one for your node. In this tutorial, you will create a config file at: `~/.avalanchego/config.json`. Note: you can create a config file anywhere on your file system, you will just need to specify its location via the flag `--config-file=<file path>` when you start your node. See [this](#) for more info on configuration file and flags.

You will need to add the DFK Subnet ID to the track Subnets section of the config file:

```
{
  <OTHER-CONFIGURATIONS>
  "track-subnets": "Vn3aX6hNRstj5VHHm63TCgPNaeGnRSqCYXQqemSqDd2TQH4qJ"
}
```

Track Subnets is a comma separated list of Subnet IDs, so if you are validating more than one Subnet, you can simply add a comma to the end of the list and append the DFK Subnet ID `Vn3aX6hNRstj5VHHm63TCgPNaeGnRSqCYXQqemSqDd2TQH4qJ`.

Running the Node

First, make sure to shut down your node in case it is still running. Then, you will navigate back into the AvalancheGo directory:

```
cd $GOPATH/src/github.com/ava-labs/avalanchego
```

If you went through the steps to set up a config file, then you can launch your node by specifying the config file on the command line:

```
./build/avalanchego --config-file ~/.avalanchego/config.json
```

If you want to track the Subnets through the command-line flag. You can append the other flags or even the `--config-file` flag as well, according to your need.

```
./build/avalanchego --track-subnets Vn3aX6hNRstj5VHHm63TCgPNaeGnRSqCYXQgemSqDd2TQH4qJ
```

Just Want the Commands? We Got You

:::caution Run `go version`. It should be 1.19.6 or above. Run `echo $GOPATH`. It should not be empty. :::

```
mkdir -p $GOPATH/src/github.com/ava-labs
cd $GOPATH/src/github.com/ava-labs
git clone https://github.com/ava-labs/avalanchego.git
cd avalanchego
./scripts/build.sh
cd $GOPATH/src/github.com/ava-labs
git clone https://github.com/ava-labs/subnet-evm.git
cd subnet-evm
./scripts/build.sh ~/.avalanchego/plugins/mDV3QWRXfwgKUWb9sggkv4vQxAQR4y2CyKrt5pLZ5SzQ7EHBv
cd $GOPATH/src/github.com/ava-labs/avalanchego
./build/avalanchego --track-subnets Vn3aX6hNRstj5VHHm63TCgPNaeGnRSqCYXQgemSqDd2TQH4qJ
```

:::info The repository cloning method used is HTTPS, but SSH can be used too:

```
git clone git@github.com:ava-labs/avalanchego.git
git clone git@github.com:ava-labs/subnet-evm.git
```

You can find more about SSH and how to use it [here](#). :::

What's the Subnet Development Lifecycle?

As you begin your Subnet journey, it's useful to look at the lifecycle of taking a Subnet from idea to production.

Figure Out Your Needs

The first step of planning your Subnet is determining your applications needs. What features do you need that the Avalanche C-Chain doesn't provide? Perhaps you want your own gas token or only want to allow access to KYCed customers. [When to Build on a Subnet vs. on the C-Chain](#) can help you make the decision.

Decide What Type of Subnet You Want

Once you've decided to use a Subnet, you need to decide what type of Subnet to build. This means choosing a virtual machine (VM) to create your Subnet with. Broadly speaking, there are three types of VMs to choose from:

EVM-Based Subnets

EVM-based Subnets are forks of the Avalanche C-Chain. They support Solidity smart contracts and standard [Ethereum APIs](#). [Subnet-EVM](#) is Ava Labs' implementation of an EVM-Based Subnet.

Subnet-EVM is far and away the safest and most popular choice to build your Subnet with. It has the most mature developer tooling and receives regular updates by the Ava Labs team.

Experimental Subnets

Experimental Subnets are proof-of-concept VMs developed by Ava Labs. They include the [TimestampVM Go](#), [TimestampVM Rust](#), and others. These VMs are demo software and aren't ready for production environments. Although they do receive periodic updates, the Ava Labs team hasn't audited their performance and security, internally or externally. However, these open source projects are intended to inspire the community, and provide novel capabilities not available inside the EVM.

If you're looking to push the boundaries of what's possible with Subnets, this is a great place to get started.

Custom Subnets

Custom Subnets are an open-ended interface that allow developers to build any VM they can dream. These VMs may be a fork of an existing VM such as Subnet-EVM, or even a non-Avalanche-native VM such as Solana's virtual machine. Alternatively, you can build your VM entirely from scratch using almost any programming language. See [Introduction to VMs](#) for advice on getting started.

Determine Tokenomics

Subnets are powered by gas tokens. When you build a Subnet, you have the option to decide what token you use and optionally, how you distribute it. You may use AVAX, an existing C-Chain token, or a brand new token. You'll need to decide how much of the supply you want to use to reward validators, what kind of emitting schedule you want, and how much to airdrop. Blocks may burn transaction fees or give them to validators as a block reward.

Decide how to Customize Your Subnet

After you've selected your VM, you may want to customize it. This may mean airdropping tokens to your team in the genesis, setting how gas fees rates on your network, or changing the behavior of your VM via precompiles. These customizations are hard to get right on paper and usually require some trial and error to determine correctly. See [Customize Your EVM-Powered Subnet](#) for instructions on configuring Subnet-EVM.

Learn Avalanche-CLI

Now that you've specified your Subnet requirements, the next step is learning Avalanche-CLI.

Avalanche-CLI is the best tool for rapidly prototyping Subnets and deploying them to production. You can use it throughout the entire Subnet development lifecycle. To get started, take a look at [Build Your First Subnet](#).

Deploy Your Subnet Locally

The first stage of Subnet development involves testing your Subnet on your local machine or on a private cloud server such as Amazon EC2. The goal of this phase is to lock-in your Subnet customizations and create your full-stack dapp without the constraints of deploying on a public network.

Development is extremely quick and updates take seconds to minutes to apply. In this phase, you should restrict dapp access to trusted parties. Because you're interacting with a local copy of the Avalanche network, you can't access production state or other production Subnets.

Deploy Your Subnet to Fuji

The second stage of Subnet development involves deploying your Subnet and your dapp to the Fuji Testnet. This phase tests your ability to run validator nodes, coordinate all parties involved in the Subnet, and monitor network health. You can also practice using Ledgers to manage Subnet transactions.

The Subnet is publicly visible and you may want to allow external users to test your dapp. Development on Fuji is significantly slower than with local development. Updates may now take hours to days to apply. It's important to do as much local testing as possible before moving to Fuji.

Deploy Your Subnet to Mainnet

The final stage of Subnet development is creating your Subnet on Mainnet and deploying your dapp. It's time to let your real users in.

Once your Subnet is in production, you have little flexibility to change it. Some alterations are possible, but they require days to weeks to implement and roll out.

Your focus should shift to preserving network stability and upgrading your nodes as needed.

Start Developing Custom VMs

If you've mastered Subnet-EVM and are looking for an additional challenge, consider building a custom VM. The Avalanche network supports far more than just the EVM. Current VMs only scratch the surface of what's possible.

Some ideas:

- Port an existing blockchain project to Avalanche. For example: Use Bitcoin's VM or Solana's VM.
- Create an app-specific VM instead of a general purpose VM. For example, create a DEX or a CLOB VM instead of something like the EVM.
- Create a more efficient implementation of the EVM.

Upgrade a Subnet

In the course of Subnet operation, you will inevitably need to upgrade or change some part of the software stack that is running your Subnet. If nothing else, you will have to upgrade the AvalancheGo node client. Same goes for the VM plugin binary that is used to run the blockchain on your Subnet, which is most likely the [Subnet-EVM](#), the Subnet implementation of the Ethereum virtual machine.

Node and VM upgrades usually don't change the way your Subnet functions, instead they keep your Subnet in sync with the rest of the network, bringing security, performance and feature upgrades. Most upgrades are optional, but all of them are recommended, and you should make optional upgrades part of your routine Subnet maintenance. Some upgrades will be mandatory, and those will be clearly communicated as such ahead of time, you need to pay special attention to those.

Besides the upgrades due to new releases, you also may want to change the configuration of the VM, to alter the way Subnet runs, for various business or operational needs. These upgrades are solely the purview of your team, and you have complete control over the timing of their roll out. Any such change represents a **network upgrade** and needs to be carefully planned and executed.

:::warning Network Upgrades Permanently Change the Rules of Your Subnet.

Procedural mistakes or a botched upgrade can halt your Subnet or lead to data loss!

When performing a Subnet upgrade, every single validator on the Subnet will need to perform the identical upgrade. If you are coordinating a network upgrade, you must schedule advance notice to every Subnet validator so that they have time to perform the upgrade prior to activation. Make sure you have direct line of communication to all your validators! :::

This tutorial will guide you through the process of doing various Subnet upgrades and changes. We will point out things to watch out for and precautions you need to be mindful about.

General Upgrade Considerations

When operating a Subnet, you should always keep in mind that Proof of Stake networks like Avalanche can only make progress if sufficient amount of validating nodes are connected and processing transactions. Each validator on a Subnet is assigned a certain `weight`, which is a numerical value representing the significance of the node in consensus decisions. On the Primary Network, weight is equal to the amount of AVAX staked on the node. On Subnets, weight is currently assigned by the Subnet owners when they issue the transaction [adding a validator](#) to the Subnet.

Subnets can operate normally only if validators representing 80% or more of the cumulative validator weight is connected. If the amount of connected stake falls close to or below 80%, Subnet performance (time to finality) will suffer, and ultimately the Subnet will halt (stop processing transactions).

You as a Subnet operator need to ensure that whatever you do, at least 80% of the validators' cumulative weight is connected and working at all times.

:::info

It is mandatory that the cumulative weight of all validators in the Subnet must be at least the value of `snow-sample-size` (default 20). For example, if there is only one validator in the Subnet, its weight must be at least `snow-sample-size`. Hence, when assigning weight to the nodes, always use values greater than 20. Recall that a validator's weight can't be changed while it is validating, so take care to use an appropriate value.

:::

Upgrading Subnet Validator Nodes

AvalancheGo, the node client that is running the Avalanche validators is under constant and rapid development. New versions come out often (roughly every two weeks), bringing added capabilities, performance improvements or security fixes. Updates are usually optional, but from time to time (much less frequently than regular updates) there will be an update that includes a mandatory network upgrade. Those upgrades are **MANDATORY** for every node running the Subnet. Any node that does not perform the update before the activation timestamp will immediately stop working when the upgrade activates.

That's why having a node upgrade strategy is absolutely vital, and you should always update to the latest AvalancheGo client immediately when it is made available.

For a general guide on upgrading AvalancheGo check out [this tutorial](#). When upgrading Subnet nodes and keeping in mind the previous section, make sure to stagger node upgrades and start a new upgrade only once the previous node has successfully upgraded. Use the [Health API](#) to check that `healthy` value in the response is `true` on the upgraded node, and on other Subnet validators check that `platform.getCurrentValidators()` has `true` in `connected` attribute for the upgraded node's `nodeID`. Once those two conditions are satisfied, node is confirmed to be online and validating the Subnet and you can start upgrading another node.

Continue the upgrade cycle until all the Subnet nodes are upgraded.

Upgrading Subnet VM Plugin Binaries

Besides the AvalancheGo client itself, new versions get released for the VM binaries that run the blockchains on the Subnet. On most Subnets, that is the [Subnet-EVM](#), so this tutorial will go through the steps for updating the `subnet-evm` binary. The update process will be similar for updating any VM plugin binary.

All the considerations for doing staggered node upgrades as discussed in previous section are valid for VM upgrades as well.

In the future, VM upgrades will be handled by the [Avalanche-CLI tool](#), but for now we need to do it manually.

Go to the [releases page](#) of the Subnet-EVM repository. Locate the latest version, and copy link that corresponds to the OS and architecture of the machine the node is running on (`darwin` = Mac, `amd64` = Intel/AMD processor, `arm64` = Arm processor). Log into the machine where the node is running and download the archive, using `wget` and the link to the archive, like this:

```
wget https://github.com/ava-labs/subnet-evm/releases/download/v0.2.9/subnet-evm_0.2.9_linux_amd64.tar.gz
```

This will download the archive to the machine. Unpack it like this (use the correct filename, of course):

```
tar xvf subnet-evm_0.2.9_linux_amd64.tar.gz
```

This will unpack and place the contents of the archive in the current directory, file `subnet-evm` is the plugin binary. You need to stop the node now (if the node is running as a service, use `sudo systemctl stop avalanchego` command). You need to place that file into the `plugins` directory where the AvalancheGo binary is located. If the node is installed using the `install` script, the path will be `~/avalanche-node/plugins`. Instead of the `subnet-evm` filename, VM binary needs to be named as the VM ID of the chain on the Subnet. For example, for the [WAGMI Subnet](#) that VM ID is `srEXiWaHuhNyGwPUi444Tu47ZEDwxTwrbQiuD7FmgSAQ6X7Dy`. So, the command to copy the new plugin binary would look like:

```
cp subnet-evm ~/avalanche-node/plugins/srEXiWaHuhNyGwPUi444Tu47ZEDwxTwrbQiuD7FmgSAQ6X7Dy
```

:::warning Make sure you use the correct VM ID, otherwise, your VM will not get updated and your Subnet may halt. :::

After you do that, you can start the node back up (if running as service do `sudo systemctl start avalanchego`). You can monitor the log output on the node to check that everything is OK, or you can use the [info.getNodeVersion\(\)](#) API to check the versions. Example output would look like:

```
{
  "jsonrpc": "2.0",
  "result": {
    "version": "avalanche/1.7.18",
    "databaseVersion": "v1.4.5",
    "gitCommit": "b6d5827f1a87e26da649f932ad649a4ea0e429c4",
    "vmVersions": {
      "avm": "v1.7.18",
      "evm": "v0.8.15",
      "platform": "v1.7.18",
      "sqja3uK17MjxfC7AN8nGadBw9JK5BcrsNwNynsqP5Gih8M5Bm": "v0.0.7",
      "srEXiWaHuhNyGwPUi444Tu47ZEDwxTwrbQiuD7FmgSAQ6X7Dy": "v0.2.9"
    }
  },
  "id": 1
}
```

Note that entry next to the VM ID we upgraded correctly says `v0.2.9`. You have successfully upgraded the VM!

Refer to the previous section on how to make sure node is healthy and connected before moving on to upgrading the next Subnet validator.

If you don't get the expected result, you can stop the `AvalancheGo`, examine and follow closely step-by-step of the above. You are free to remove files under `~/avalanche-node/plugins`, however, you should keep in mind that removing files is to remove an existing VM binary. You must put the correct VM plugin in place before you restart `AvalancheGo`.

Network Upgrades

Sometimes you need to do a network upgrade to change the configured rules in the genesis under which the Chain operates. In regular EVM, network upgrades are a pretty involved process that includes deploying the new EVM binary, coordinating the timed upgrade and deploying changes to the nodes. But since [Subnet-EVM v0.2.8](#), we introduced the long awaited feature to perform network upgrades by just using a few lines of JSON. Upgrades can consist of enabling/disabling particular precompiles, or changing their parameters. Currently available precompiles allow you to:

- Restrict Smart Contract Deployers
- Restrict Who Can Submit Transactions
- Mint Native Coins
- Configure Dynamic Fees

Please refer to [Customize a Subnet](#) for a detailed discussion of possible precompile upgrade parameters.

Summary

Vital part of Subnet maintenance is performing timely upgrades at all levels of the software stack running your Subnet. We hope this tutorial will give you enough information and context to allow you to do those upgrades with confidence and ease. If you have additional questions or any issues, please reach out to us on [Discord](#).

Troubleshooting Subnet Deployments

If you run into trouble deploying your Subnet, use this document for tips to resolve common issues.

Deployment Times Out

During a local deployment, your network may fail to start. Your error may look something like this:

```
[~]$ avalanche subnet deploy mySubnet
✓ Local Network
Deploying [mySubnet] to Local Network
Backend controller started, pid: 26388, output at: /Users/user/.avalanche-cli/runs/server_20221231_111605/avalanche-cli-backend
VMs ready.
Starting network...
.....
.....Error: failed to query network health: rpc error: code = DeadlineExceeded desc = context deadline exceeded
```

Avalanche-CLI only supports running one local Avalanche network at a time. If other instances of `AvalancheGo` are running concurrently, your Avalanche-CLI network fails to start.

To test for this error, start by shutting down any Avalanche nodes started by Avalanche-CLI.

```
avalanche network clean --hard
```

Next, look for any lingering `AvalancheGo` processes with:

```
ps aux | grep avalanchego
```

If any processes are running, you need to stop them before you can launch your VM with Avalanche-CLI.

:::warning

If you're running a validator node on the same box you're using Avalanche-CLI, **don't** end any of these lingering `AvalancheGo` processes. This may shut down your validator and could affect your validation uptime.

:::

Incompatible RPC Version for Custom VM

If you're locally deploying a custom VM, you may run into this error message.

```
[~]$ avalanche subnet deploy mySubnet
✓ Local Network
Deploying [mySubnet] to Local Network
Backend controller started, pid: 26388, output at: /Users/user/.avalanche-cli/runs/server_20221231_111605/avalanche-cli-backend
VMs ready.
Starting network...
.....
Blockchain has been deployed. Wait until network acknowledges...
```

```
.....Error: failed to query network health: rpc error: code = DeadlineExceeded desc = context deadline exceeded
```

This error has many possible causes, however you should be aware of the possibility of an RPC protocol version mismatch.

AvalancheGo communicates with custom VMs over RPC using [gRPC](#). gRPC defines a protocol specification shared by both AvalancheGo and the VM. Both components **must** be running the same RPC version for VM deployment to work. You can view [AvalancheGo's RPC compatibility broken down by release version](#).

Your custom VM's RPC version is set by the version of AvalancheGo that you import. By default, Avalanche-CLI creates local Avalanche networks that run the latest AvalancheGo release.

Here's an example with real numbers from the AvalancheGo compatibility page. If the latest AvalancheGo release is version v1.9.4, then Avalanche-CLI deploys a network with RPC version 20. For your deploy to be successful, your VM must also have RPC version 20. Because only AvalancheGo version v1.9.4 supports RPC version 20, your VM **must** import AvalancheGo version v1.9.4. If the latest release were v1.9.3, your VM could safely import versions v1.9.2 or v1.9.3 because both versions support RPC version 19.

If your VM has an RPC version mismatch, you have two options. First, you can update the version of AvalancheGo you use in your VM. This is the correct long-term approach. However, there is a short-term workaround. Avalanche-CLI supports deploying older versions of AvalancheGo by using the `--avalanchego-version` flag. Both the `subnet_deploy` and `network_start` commands support setting the AvalancheGo version explicitly. Although it's very important to keep your version of AvalancheGo up-to-date, this workaround helps you avoid broken builds in the short term. However, you need to upgrade to the latest AvalancheGo version when deploying publicly to the Fuji Testnet or Mainnet.

:::note Updates to AvalancheGo's RPC version are not tied to its semantic version scheme. Minor AvalancheGo version bumps may include a breaking RPC version bump. :::

sidebar_position: 7

WAGMI Subnet Demo

The WAGMI ("We're All Going to Make It") Subnet Demo is a high throughput testbed for EVM (Ethereum Virtual Machine) optimizations. It is parameterized to run at a factor more capacity than Fuji/Mainnet C-Chain and will be used to experiment with release candidates before they make it into an official Coreth release.

Network Parameters

- NetworkID: 11111
- ChainID: 11111
- Block Gas Limit: 20,000,000 (2.5x C-Chain)
- 10s Gas Target: 100,000,000 (~6.67x C-Chain)
- Min Fee: 1 Gwei (4% of C-Chain)
- Target Block Rate: 2s (Same as C-Chain)

Genesis file of WAGMI can be found [here](#).

Everyone that has used the C-Chain more than twice (~970k addresses) has been airdropped 10 WGM tokens. With the current fee parameterization, this should be enough for hundreds of TXs.

This is one of the first cases of using Avalanche Subnets as a proving ground for changes in a production VM (Coreth). Many underestimate how useful the isolation of Subnets is for performing complex VM testing on a live network (without impacting the stability of the primary network).

We created a basic WAGMI Explorer <https://subnets-test.avax.network/wagmi> that surfaces aggregated usage statistics about the Subnet.

Subnet Info

- SubnetID: [28nrH5T2BMvNrWecFcV3mfccjs6axM1TVyge79MCv2Mhs8kxiY](https://subnets-test.avax.network/wagmi)
- ChainID: [2ebCneCbwhlQ1rYT41nhd7M76Hc6YmosMAQrTFhBq8qeqjh6tt](https://subnets-test.avax.network/wagmi)

Adding WAGMI to MetaMask

- Network Name: WAGMI
- RPC URL: <<https://subnets-test.avax.network/wagmi/wagmi-chain-testnet/rpc>>
- WS URL: [wss://subnets-test.avax.network/wagmi/wagmi-chain-testnet/ws](https://subnets-test.avax.network/wagmi/wagmi-chain-testnet/ws)
- Chain ID: 11111
- Symbol: WGM
- Explorer: <<https://subnets-test.avax.network/wagmi/wagmi-chain-testnet/explorer>>

When to Build on a Subnet vs. on the C-Chain

A Subnet is a subset of Avalanche Primary Network validators agreeing to run the same [Virtual Machines \(VM\)](#) with its own rules. Subnet enables extra dimensions of reliability, efficiency, and data sovereignty. It provides the ability to create custom blockchains for different use cases, while isolating high-traffic applications from congesting activity on the Primary Network. But such flexibility comes with its own set of tradeoffs. In this article, we discuss often-overlooked differentiating characteristics of Subnet, with primary focus on EVM-based applications (for example, C-Chain, [Subnet-EVM](#)). The goal is to identify pros and cons of building an app on C-Chain versus [Subnet-EVM](#), and help developers make more informed decisions.

When to Use a Subnet

There are many advantages to running your own Subnet. If you find one or more of these a good match for your project then a Subnet might be a good solution for you. But make sure to check the reasons to use the C-Chain instead, as some trade-offs involved might make that a preferred solution.

We Want Our Own Gas Token

C-Chain is an Ethereum Virtual Machine (EVM) chain; it requires the gas fees to be paid in its native token. That is, the application may create its own utility tokens (ERC-20) on the C-Chain, but the gas must be paid in AVAX. In the meantime, [Subnet-EVM](#) effectively creates an application-specific EVM-chain with full control over native(gas) coins. The operator can pre-allocate the native tokens in the chain genesis, and mint more using the [Subnet-EVM](#) precompile contract. And these fees can be either burned (as AVAX burns in C-Chain) or configured to be sent to an address which can be a smart contract.

Note that the Subnet gas token is specific to the application in the chain, thus unknown to the external parties. Moving assets to other chains requires trusted bridge contracts (or upcoming cross Subnet communication feature).

We Want Higher Throughput

The primary goal of the gas limit on C-Chain is to restrict the block size and therefore prevent network saturation. If a block can be arbitrarily large, it takes longer to propagate, potentially degrading the network performance. The C-Chain gas limit acts as a deterrent against any system abuse but can be quite limiting for high throughput applications. Unlike C-Chain, Subnet can be single-tenant, dedicated to the specific application, and thus host its own set of validators with higher bandwidth requirements, which allows for a higher gas limit thus higher transaction throughput. Plus, [Subnet-EVM](#) supports fee configuration upgrades that can be adaptive to the surge in application traffic.

Subnet workloads are isolated from the Primary Network; which means, the noisy neighbor effect of one workload (for example NFT mint on C-Chain) cannot destabilize the Subnet or surge its gas price. This failure isolation model in the Subnet can provide higher application reliability.

We Want Strict Access Control

The C-Chain is open and permissionless where anyone can deploy and interact with contracts. However, for regulatory reasons, some applications may need a consistent access control mechanism for all on-chain transactions. With [Subnet-EVM](#), an application can require that "only authorized users may deploy contracts or make transactions." Allow-lists are only updated by the administrators, and the allow list itself is implemented within the precompile contract, thus more transparent and auditable for compliance matters.

We Need EVM Customization

If your project is deployed on the C-Chain then your execution environment is dictated by the setup of the C-Chain. Changing any of the execution parameters means that the configuration of the C-Chain would need to change, and that is expensive, complex and difficult to change. So if your project needs some other capabilities, different execution parameters or precompiles that C-Chain does not provide, then Subnets are a solution you need. You can configure the EVM in a Subnet to run however you want, adding precompiles, and setting runtime parameters to whatever your project needs.

When to Use the C-Chain

All the reasons for using a Subnet outlined above are very attractive to developers and might make it seem that every new project should look into launching a Subnet instead of using the C-Chain. Of course, things are rarely that simple and without trade-offs. Here are some advantages of the C-Chain that you should take into account.

We Want High Composability with C-Chain Assets

C-Chain is a better option for seamless integration with existing C-Chain assets and contracts. It is easier to build a DeFi application on C-Chain, as it provides larger liquidity pools and thus allows for efficient exchange between popular assets. A DeFi Subnet can still support composability of contracts on C-Chain assets but requires some sort of off-chain system via the bridge contract. In other words, a Subnet can be a better choice if the application does not need high composability with the existing C-Chain assets. Plus, the upcoming support for cross Subnet communication will greatly simplify the bridging process.

We Want High Security

The security of Avalanche Primary Network is a function of the security of the underlying validators and stake delegators. Some choose C-Chain in order to achieve maximum security by utilizing thousands of Avalanche Primary Network validators. Some may choose to not rely on the entire security of the base chain.

The better approach is to scale up the security as the application accrues more values and adoption from its users. And Subnet can provide elastic, on-demand security to take such organic growth into account.

We Want Low Initial Cost

C-Chain has economic advantages of low-cost deployment, whereas each Subnet validator is required to validate the Primary Network by staking AVAX (minimum 2,000 AVAX for Mainnet). For fault tolerance, we recommend at least five validators for a Subnet, even though there is no requirement that the Subnet owner should own all these 5 validators, it still further increases the upfront costs.

We Want Low Operational Costs

C-Chain is run and operated by thousands of nodes, it is highly decentralized and reliable, and all the infrastructure (explorers, indexers, exchanges, bridges) has already been built out by dedicated teams that maintain them for you at no extra charge. Project deployed on the C-Chain can leverage all of that basically for free. On the other hand, if you run your own Subnet you are basically in charge of running your own L1 network. You (or someone who you partner with or pay to) will need to do all those things and you will ultimately be responsible for them. If you don't have a desire, resources or partnerships to operate a high-availability 24/7 platform, you're probably better off deploying on the C-Chain.

Conclusion

Here we presented some considerations both in favor of running your own Subnet and in favor of deploying on the C-Chain. You should carefully weigh and consider what makes the most sense for you and your project: deploying on a Subnet or deploying on the C-Chain.

But, there is also a third way: deploy on C-Chain now, then move to your own Subnet later. If an application has relatively low transaction rate and no special circumstances that would make the C-Chain a non-starter, you can begin with C-Chain deployment to leverage existing technical infrastructure, and later expand

to a Subnet. That way you can focus on working on the core of your project and once you have a solid product/market fit and have gained enough traction that the C-Chain is constricting you, plan a move to your own Subnet.

Of course, we're happy to talk to you about your architecture and help you choose the best path forward. Feel free to reach out to us on [Discord](#) or other [community channels](#) we run.